TAINT-DRIVEN FIRMWARE FUZZING

OF

EMBEDDED SYSTEMS

THESIS

Submitted in Partial Fulfillment of

the Requirements for

the Degree of

MASTER OF SCIENCE (Computer Science)

at the

NEW YORK UNIVERSITY

TANDON SCHOOL OF ENGINEERING

by

Melisa Kristi Savich

May 2020

Advisor Signature

5/15/2020

Date

Department Chair Signature

5/15/2020

Date

N13728581

University ID

mks629

Net ID

Vita

Melisa Kristi Savich was born in Detroit, Michigan on September 13th, 1995. After graduating from Stevenson High School in 2013, she began her undergraduate studies at the University of Michigan-Dearborn. She originally pursued a pre-medical track alongside a French Studies major with a Women's and Gender Studies major. In Winter 2015, she transferred to the University of Michigan. Still intrigued by language, the social and cultural constructs of gender, systems of privilege and oppression, and the relationships between power and gender, she continued her French and Women's Studies coursework. At the University of Michigan, Melisa took a Computer Science course taught by Amir Kamil to fulfill a general education requirement. This course inspired her to continue learning in the area of computer science which eventually led her to become a student instructor during her two last years at the University of Michigan. Upon graduating with a Bachelor of Science in both Computer Science and Women's Studies, she was accepted into the Critical Skills Master's Program by Sandia National Laboratories which funded her Master's degree in Computer Science at New York University's Tandon School of Engineering. This research was completed from September 2018 to May 2020.

Acknowledgments

I would like to thank my thesis advisor, Dr. Brendan Dolan-Gavitt to the greatest extent for his time, patience, suggestions, and feedback throughout the course of this research project. Without his support from the day I met him to this moment, this project would not exist in the way it does now. I am forever grateful for the weekly meetings, emails, and Slack communications regarding discussions and problems I came across while completing this project.

Special distinctions need to be made to Dr. Amir Kamil and Dr. Peter Honeyman at the University of Michigan.

The first computer science course I ever took was taught by Dr. Kamil. Being a student in his class was an extreme pleasure and gave me such a positive outlook on continuing the study of computer science outside of his class, especially as a woman in her first engineering course. Not long after I ended up becoming a student instructor for Dr. Kamil's intermediate-level course, and was welcomed by a cohort of students each with their own diverse viewpoints and strengths. My time with my fellow student instructors, Dr. Kamil, as well as the other faculty teaching this course solidified my desire to pursue a degree in computer science.

I would also like to give a special thank you to Dr. Peter Honeyman for his expertise, candidness, and friendship. The discussions I've had with him regarding my thesis over the course of these past two years have not been forgotten and have forever lingered in my head. I very much appreciated his unreserved comments towards my work because they further fueled me to work harder. Inadvertently you have also given me a space to

think about things beyond the bubble of the area we call the study of computer science. I appreciate your words and getting to know you over the course of these past few years. I'm privileged to call you my friend. Thank you.

To my parents,

Sašo and Julija Savich.

AN ABSTRACT

TAINT-DRIVEN FIRMWARE FUZZING OF EMBEDDED SYSTEMS

by

Melisa Kristi Savich

Advisor: Brendan Dolan-Gavitt, Ph.D.

Submitted in Partial Fulfillment of the Requirements

for the Degree of Master of Science (Computer Science)

May 2020

Fuzzing has become a growingly popular method for finding software bugs. Interest in this area of research has been partially reignited by DARPA's Cyber Grand Challenge (CGC) binaries, which focused on creating automatic systems capable of finding flaws and formulating patches [3]. Since then, a multitude of effective and diverse fuzzers have been published like AFLGo, Angora, Driller, QSYM, and VUzzer. Prior to the Cyber Grand Challenge, fuzzing was already an area of interest and fuzzers like BuzzFuzz and COMET had been developed. All of these fuzzers have presented powerful techniques in the area of software fuzzing.

However, embedded systems do not get to benefit from these recent advances. With such a range in the complexity and purposes of these devices, along with the fact that source code is usually unavailable, finding an efficient, correct, and easy approach to testing the security of these devices seems infeasible.

A variety of diverse approaches and techniques have come out in recent years. Methods for input generation for these fuzzers range from mutational, generational, sequential, constraint solving or some unique combination of the four. Each strategy has its own advantages and disadvantages. Through research, it's been shown that combinations of these techniques yield the best results. Fuzzers like AFLGo, Angora, BuzzFuzz, COMET, Driller, QSYM, and VUzzer have all proven to be successful in certain scenarios and have been proven to be efficient at fuzzing software. We present TDFF, a Taint-Driven Firmware Fuzzer that attempts to solve the problem of fuzzing embedded

systems at scale by making use of taint analysis paired with fuzzing strategies from afl-fuzz [18].

Table of Contents

Chapter 1

Introduction

Fuzzing has become a growingly popular method for finding software bugs. Interest in this area of research has been partially reignited by DARPA's Cyber Grand Challenge (CGC) binaries, which focused on creating automatic systems capable of finding flaws and formulating patches [3]. Since then, a multitude of effective and diverse fuzzers have been published like AFLGo, Angora, Driller, QSYM, and VUzzer. Prior to the Cyber Grand Challenge, fuzzing was already an area of interest and fuzzers like BuzzFuzz and COMET had been developed. All of these fuzzers have presented powerful techniques in the area of software fuzzing.

However, embedded systems do not get to benefit from these recent advances. With new embedded devices being put on the market every day, these systems are becoming increasingly difficult to test. Companies previously uninvolved in tech are now coming out with "smart" products for consumer homes like smart light bulbs and refrigerators. The medical field also makes use of embedded technology ranging from automated insulin pumps to smart pacemakers. Critical infrastructure systems also rely on these embedded devices – water treatment, electric power transmission, and wind farms all most commonly make use of a variety of programmable logic controllers (PLCs). With such a range in the complexity and purposes of these devices, along with the fact that source code is usually unavailable, finding an efficient, correct, and easy approach to

testing the security of these devices seems infeasible.

We attempt to address this problem by building on top of already existing projects. Avatar^2 is an orchestration framework designed to support dynamic analysis of embedded devices [8]. Avatar^2 makes use of PANDA, an open-source Platform for Architecture-Neutral Dynamic Analysis. We make use of one of PANDA's unique features – the ability to record whole system executions of embedded devices [4]. Within PANDA, we write a taint analysis plugin and use it during replays of execution recordings. This plugin helps make informed decisions of which areas of code to fuzz in embedded system firmware, helping us find meaningful bugs.

Chapter 2

Related Work

A variety of diversive fuzzers have come out in recent years. Methods for input generation for these fuzzers range from mutational, generational, sequential, constraint solving or some unique combination of the four. As research has shown, each strategy has its own advantages and disadvantages. Whilst some fuzzers are quick at generating input, others take additional time in order to create fewer, yet more quality inputs. Through research, it's been shown that a combination of both these techniques yield the best results. Fuzzers like AFLGo, Angora, BuzzFuzz, COMET, Driller, QSYM, and VUzzer have all proven to be successful in certain scenarios and have been proven to be efficient at fuzzing software.

The developers of AFLGo introduce the technique of Directed Greybox Fuzzing (DGF), which generates inputs with the objective of reaching a given set of target program locations efficiently [1]. The researchers argue that most existing directed fuzzers are based on symbolic execution, which spend considerable time with heavy-weight program analysis and constraint solving. They introduce the meta-heuristic of Simulated Annealing which assigns more energy to seeds that are closer to the target locations while reducing energy for seeds that are further away. However, they conclude that this technique is primarily useful in the areas of patch testing and crash reproduction.

While fuzzers based on symbolic execution produce quality inputs, input generation

is slow. Many fuzzers have come out in recent years which attempt to solve path constraints without symbolic execution. Angora [2] is one of them. Angora introduced a variety of techniques to solve path constraints efficiently: context-sensitive branch coverage, scalable byte-level taint tracking, search based on gradient descent, type and shape inference, and input length exploration.

BuzzFuzz uses dynamic taint tracing to automatically locate regions of original seed input files that influence values used at key program attack points [5]. The taint information identifies promising locations of the input file to fuzz, while preserving the syntactic structure of the original input file. This technique allows BuzzFuzz to expose errors located deep within large programs. However, a drawback is that the taint instrumentator takes as input the source of the program under test, which is usually unavailable for firmware images.

Another novel idea that has been introduced in the fuzzing domain is the idea of a feedback loop. COMET (COverage Maximization using Taint) was developed as a system for automatically obtaining a test suite for a program via a feedback loop that maximizes line coverage directly [6]. Each iteration of the feedback loop considers a set of inputs and for each input, a set of conditionals are both tainted by the inputs and for which only one branch is covered. For each such conditional, the algorithm searches for an input that exposes the other branch of the conditional, adding any it discovers for consideration in the next iteration.

Driller is yet another fuzzer that has been proposed in recent years. It is a hybrid vulnerability excavation tool which leverages fuzzing and selective concolic execution in a complementary manner, to find deeper bugs [17]. It leverages selective concolic execution to find deep and meaningful bugs, while simultaneously improving the scalability of concolic execution by using fuzzing to alleviate path explosion.

Hybrid fuzzing is yet another approach that has been proposed by researchers. QSYM is a fast concolic execution engine designed to support hybrid fuzzing [19]. QSYM, along with other hybrid fuzzers, combine both fuzzing and concolic execution.

The hopes are that the fuzzer will quickly explore simple branches such as `x > 42` and that the concolic execution will solve complex branches such as `x == 0xDEADBEEF`.

Another fuzzer that attempts to both be scalable and find deep and meaningful bugs is VUzzer [10]. It does not require any prior knowledge of the application or input format and utilizes an evolutionary fuzzing strategy to generate input. VUzzer implements a "smart" mutation feedback loop based on control- and data-flow application features without having to resort to less scalable symbolic execution. The researchers argue that doing more work at the front-end produces fewer but better inputs.

Given this assortment of fuzzers each with their own unique approaches, we develop TDFF, a taint-driven firmware fuzzer that uses taint analysis as a mutation feedback loop.

Chapter 3

Background

This research aims to explore the feasibility and efficiency of a taint-driven firmware fuzzer. Previous research on fuzzing strategies has shown that fuzzers utilizing taint analysis are generally slower at generating input. However, the task of fuzzing firmware on embedded devices might make the use of taint analysis worth the extra overhead when generating input. The processors found on these devices are significantly slower than the ones used for smartphones, laptops, and PCs. Therefore, these devices do not have the luxury of running many test cases within a certain timeframe. We hope that the amount of time it takes to generate quality test cases is a favorable tradeoff when fuzzing on embedded devices.

3.1 Emulation

We chose the avatar^2 orchestration framework in conjunction with PANDA, the Platform for Architecture-Neutral Dynamic Analysis, as the foundation of our research.

We leverage avatar^2's ability to automatically transfer the internal state of a device/application, as well as its ability to configure the forwarding of input/output and memory accesses to physical peripherals or emulated targets [8]. This is particularly useful as we can transfer state to avatar^2 during execution we wish to analyze, and transfer the state back to the hardware when physical peripherals need to be accessed.

Avatar^2's integration of PANDA makes the choice to use avatar^2 even more favorable. PANDA's record and replay capabilities allow us to use and develop taint analysis plugins that assist in analyzing previously-recorded executions. The results from taint analysis will guide our fuzzer into generating quality inputs.

3.2 Dynamic Taint Analysis

Dynamic analysis is one of the tools researchers use in their arsenal to analyze programs. Dynamic analysis is the ability to monitor code as it executes, and is a favorable technique as it allows a researcher to examine actual executions and make determinations based on run-time information.

The use of taint information while performing dynamic analysis allows a researcher to evaluate which computations depend on taint sources. These taint sources generally come from input.

The use of dynamic taint analysis predates to 2010 and has been used in the following scenarios [16]:

- Unknown Vulnerability Detection
- Automatic Input Filter Generation
- Malware Analysis
- Test Case Generation

TDFF, the taint-driven firmware fuzzer, makes use of dynamic taint analysis in the last scenario specified; test case generation.

Chapter 4

Design and Implementation

4.1 Overview

TDFF was implemented in both C code and Python code. TDFF utilizes a three-step approach to addressing the problem at hand. The steps are as follows:

- Record
- Replay
- Explore

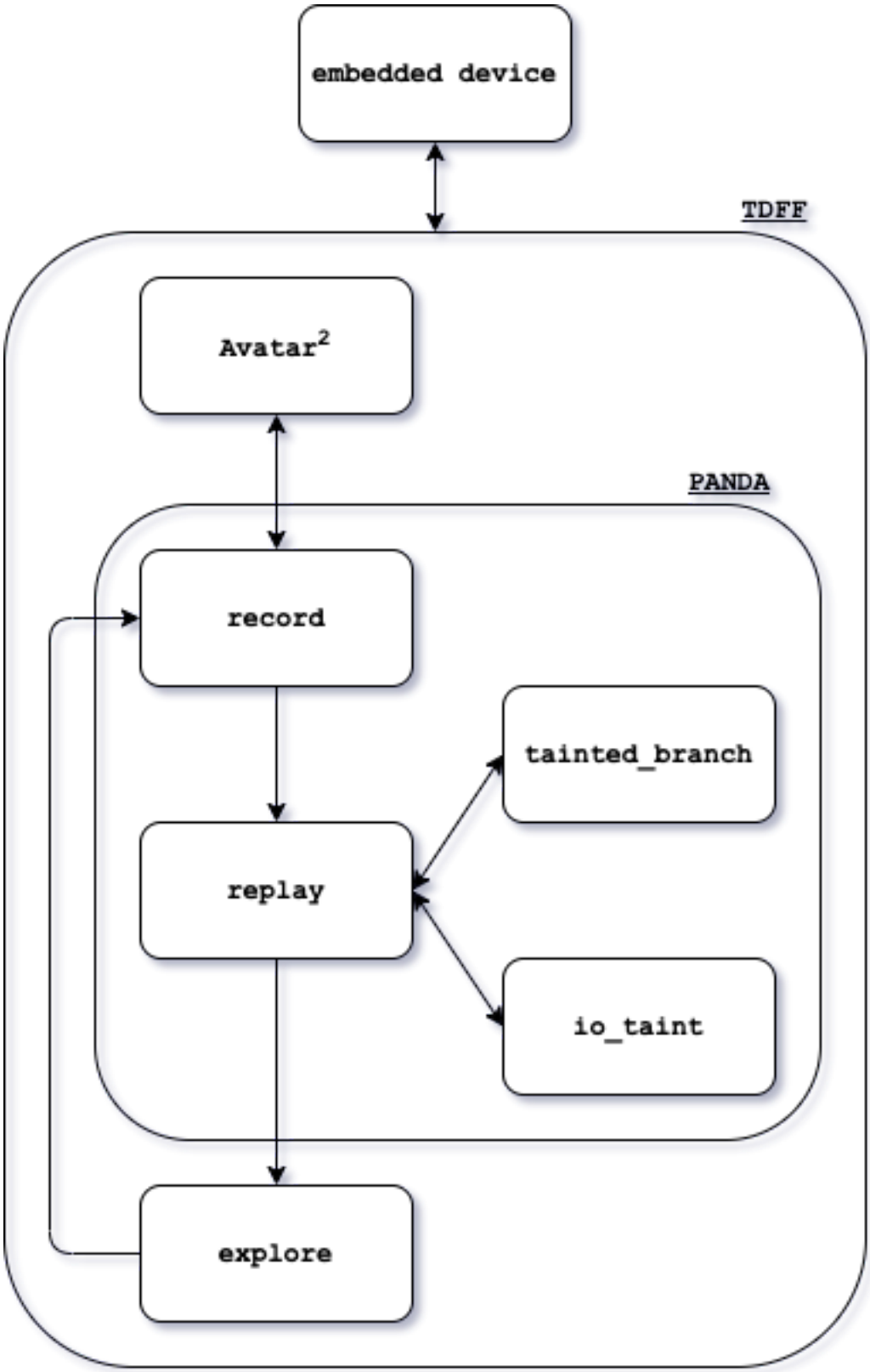

Figure 4.1: General Overview of TDFF

4.2 Setup

In order to set up TDFF, a few things need to be arranged:

- A device capable of being emulated by QEMU must be selected. We limited the devices in this paper to ARM Cortex-M3 processors. However, with modifications to TDFF, devices with a JTAG interface may be used.
- A proper transition address must be selected for the PANDA recording script. Usually this is the address of where `main()` begins. One can find such an address with the command `arm-none-eabi-objdump -d firmware.elf | grep main`.
- Both the MMIO start and end addresses must be specified in TDFF's config file. These addresses can be found in the memory map of the selected device.
- A proper exit address must be selected for the `io_taint` PANDA plugin. Usually this is the address of where `exit()` begins. One can find such an address with the command `arm-none-eabi-objdump -d firmware.elf | grep exit`.

More detailed information on how to set up and use TDFF can be found on the Github repository: `https://github.com/m3lixir-nyu/tdff` [15].

4.3 Record

A PANDA recording script must be supplied by the user. This script is responsible for capturing firmware execution that later on will be taint analysed. The user is responsible for determining which parts of firmware execution seem promising (usually this is input parsing and processing).

4.4 Replay

The PANDA recording is replayed and analysed with the `tainted_branch` and `io_taint` PANDA plugins. The `tainted_branch` plugin produces a report in pandalog (plog) format that lists the addresses of every branch instruction in the replay that depends on tainted data [4]. The `io_taint` plugin was developed as part of this project. It consists of five callbacks/features:

- `PANDA_CB_BEFORE_BLOCK_EXEC`
  - Before executing each basic block, an entry is written to the pandalog containing the address/pc of the block.
- `PANDA_CB_PHYS_MEM_BEFORE_READ`
  - Before a read of physical memory, the physical memory address is written to standard output.
- `PANDA_CB_PHYS_MEM_AFTER_READ`
  - After a read of physical memory, the bytes that were read are written to standard output in hex and ascii format. A taint label is also applied to the input.
- `PANDA_CB_PHYS_MEM_BEFORE_WRITE`
  - Before a write to physical memory, the physical memory address is written to standard output.
- `PANDA_CB_PHYS_MEM_AFTER_WRITE`
  - After a write to physical memory, the bytes that were written are written to standard output in hex and ascii format.

The resulting pandalog provides insight to TDFF about which areas of the input file to mutate on future fuzz runs.

4.5 Explore

TDFF attempts to attain two goals: 1) achieve a higher degree of code coverage and 2) discover bugs. The PANDA plugins used for analysis directly contribute to the achievement of these two goals.

During the explore phase, a given pandalog is examined. The pandalog is examined entry by entry. For each entry, TDFF checks whether the entry contains a `tainted_branch` and/or `io_taint` attribute.

If a `tainted_branch` attribute exists in the pandalog entry, it means that the entry refers to a branch instruction that depends on tainted data. TDFF creates a pair of the address of the branch instruction as well as the taint label it depends on. The pair is then examined to determine if it has been seen before. If it hasn't, TDFF chooses to explore the branch by mutating the input file at the offset that is associated with the taint label. If it has, TDFF discards the input file and does not use it as a basis for mutation in future fuzz runs.

If an `io_taint` attribute exists in the pandalog entry, it means that the entry refers to a basic block that was executed. For each entry that contains an `io_taint` attribute, TDFF records the address of the basic block. After examining the entire pandalog, TDFF reports to the user which new blocks were discovered during the most recent iteration of fuzzing.

Both of these PANDA plugins assist TDFF in achieving its goals. The `tainted_branch` plugin advises TDFF on what input files seem promising to mutate, and which areas of said input files should be mutated. The `io_taint` plugin reports the degree of code coverage in terms of basic blocks executed.

4.5.1 Fuzzing Strategies

TDFF employs the deterministic fuzzing strategies used by afl-fuzz as described in the technical whitepaper (with slight tweaks/modifications) [18]:

- sequential bit flips with varying lengths and stepovers
- sequential addition and subtraction of small integers
- sequential insertion of known interesting integers (0, 1, INT_MAX, etc)

Chapter 5

Evaluation

We tested the feasibility and efficiency of TDFF on three separate programs, a simple C program, a complex C program, and Libtasn1. The results are as follows:

5.1 Simple C Program

As a proof of concept, we created a simple C program to show the correctness of TDFF. We flashed both the Nucleo-L152RE and the Nucleo-F207ZG with the following program:

```
#include "mbed.h"

Serial pc(SERIAL_TX, SERIAL_RX);

DigitalOut myled(LED1);

int main() {
    pc.printf("What's the meaning of Life, the Universe, and Everything?\r\n");
    myled = !myled;

    char input[10];
    gets(input);

    pc.printf("You entered \"%s\".\r\n", input);

```

```
16      if (input[1] == '2' && input[0] == '4') {
17          pc.printf("Correct.\r\n");
18      } else {
19          pc.printf("Incorrect.\r\n");
20      }
21
22      return 0;
23  }
```

Listing 5.1: Simple C Program

The program simulates a situation where the correct magic bytes are needed to enter a branch of targeted code.

Given an input of:

\x34\x00\x0a

This program in particular searches for an input of:

\x34\x32\x0a

It took 18.11 minutes for TDFF to find the correct input on the Nucleo-F207ZG and a total of 5149.81 seconds (1.43 hours) to complete all fuzz runs. Due to the COVID-19 pandemic, testing was unable to be done on the Nucleo-L152RE.

5.2 Complex C Program

To showcase TDFF's ability to enter deeper parts of a program, we tested it on a slightly more complex program inspired by Listing 3 in the VUzzer paper [10]:

```
1   #include "mbed.h"
2
3   Serial pc(SERIAL_TX, SERIAL_RX);
4
5   int main () {
6       char buf[20];
7       gets(buf);
```

```
8
9       if (buf[1] == 0xEF && buf[0] == 0xFD) {
10          pc.printf("Passed first check.\r\n");
11      } else {
12          pc.printf("Failed first check.\r\n");
13          exit(1);
14      }
15
16      if (buf[10] == '%' && buf[11] == '@') {
17          pc.printf("Passed second check.\r\n");
18
19          if (strncmp((const char *) &buf[15], "MAZE", 4) == 0) {
20              pc.printf("Passed third check.\r\n");
21          } else {
22              pc.printf("Failed third check.\r\n");
23              exit(1);
24          }
25      } else {
26          pc.printf("Failed second check.\r\n");
27          exit(1);
28      }
29
30      return 0;
31  }
```

Listing 5.2: Complex C Program

Given an input of:

\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00

\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0a

This program in particular searches for an input of:

\xFD\xEF\x00\x00\x00\x00\x00\x00\x00\x00

\x25\x40\x00\x00\x00\x4D\x41\x5A\x45\x0a

For this particular program, TDFF could not find the correct input on the Nucleo-F207ZG.

5.3 Libtasn1

Libtasn1 is the ASN.1 library used by GnuTLS, p11-kit and some other packages [7]. We tested the effectiveness of TDFF on a Nucleo-F207ZG against a known buggy version of Libtasn1 (version 3.5). The source code used to flash the Nucleo can be found on Github {m3lixir-nyu-tdff.

According to NIST [9], 8 CVE's have been assigned to Libtasn1 (version 3.5). After 24 hours of running TDFF, TDFF was not able to discover even one known CVE. However, after 12 hours of running TDFF, TDFF was able to discover 1865 basic blocks.

One apparent issue that came up when evaluating the feasibility of TDFF on Libtasn1 is the rate at which data is written to the embedded device. In our example, a certificate of 699 bytes needed to be written to the program in order to begin. Through testing, we found that at the very least, a sleep of 1.5 seconds was needed between writes in order to avoid data loss through the serial port. In this particular example, this means that 1048.5 seconds need to be allocated for every run of TDFF just for the purpose of writing data. This amounts to half an hour per run of just writing data. For TDFF to scale to much larger programs that require more data than 699 bytes, another approach must be considered.

Limitations and Future Work

In this paper we've laid the foundation for fuzzing firmware on embedded systems. However, over the course of the development of TDFF, many limitations were encountered.

Since TDFF's foundation relies on avatar^2 (and therefore QEMU and PANDA), the use of TDFF is limited to firmware that is capable of being emulated by QEMU. The scope of this paper was limited to ARM devices. Because of this decision, we had discovered that avatar^2 is capable of emulating ARM Cortex-M3 devices with ease. However, since ARM Cortex-M4 support is continually being developed by QEMU, its integration with avatar^2 isn't as stable, making TDFF not as user-friendly for ARM Cortex-M4 devices.

Furthermore, avatar^2 is developed and maintained by Eurecom's S3 Group. Therefore, any bugs and/or problems encountered in avatar^2 are not as likely to be addressed unless they are already a priority of the avatar^2 developers.

Finally, as is a common problem with most other fuzzers, TDFF requires a copy of the firmware in order to properly emulate the device on avatar^2. This firmware is needed in order to properly execute state transfers between avatar^2 and the embedded device.

Conclusion

The contribution of TDFF, the Taint-Driven Firmware Fuzzer, has laid a solid foundation for the technique of taint-driven fuzzing for embedded systems. This preliminary work shows that TDFF currently works on small examples but needs further development to scale to larger real-world firmware.

We have attempted an approach to evaluating the security of embedded systems through firmware fuzzing driven by taint analysis. The system presented in this paper can be improved by extending on the development of the foundations it lies on; QEMU and PANDA. An example of such progress is shown with the development of the `io_taint` PANDA plugin created for this project. With similar developments, TDFF will have the ability to fuzz ARM Cortex-M3 and ARM Cortex-M4 devices with ease, if not others.

In addition, to increase TDFF's efficiency in fuzzing such systems, more formal and developed fuzzing strategies can be employed instead of the basic, straight-forward ones used in TDFF's current implementation.

The source code for this entire project can be found at `https://github.com/m3lixir-nyu/avatar2`, `https://github.com/m3lixir-nyu/avatar-qemu`, `https://github.com/m3lixir-nyu/avatar-panda`, and `https://github.com/m3lixir-nyu/io_taint` [11] [12] [13] [14].

Bibliography

[1] Böhme, M., Pham, V., Nguyen, M., Roychoudhury, A. *Directed Greybox Fuzzing*, 2017.

[2] Chen, P., Chen, H. *Angora: Efficient Fuzzing by Principled Search*, 2018.

[3] *Cyber Grand Challenge (CGC)*, `https://www.darpa.mil/program/cyber-grand-challenge`

[4] Dolan-Gavitt, B., Hodosh, J., Hulin, P., Leek, T., Whelan, R. *Repeatable Reverse Engineering with PANDA*, 2015.

[5] Ganesh, V., Leek, T., Rinard, M. *Taint-based Directed Whitebox Fuzzing*, 2009.

[6] Leek, T., Baker, G., Brown, Ruben., Zhivich, M., Lippmann, R. *Coverage Maximization Using Dynamic Taint Tracing*, 2007.

[7] *Libtasn1*, `https://www.gnu.org/software/libtasn1/`

[8] Muench, M., Nisi, D., Francillon, A., Balzarotti, D. *Avatar^2: A Multi-target Orchestration Platform*, 2018.

[9] *NIST: National Institute of Standards and Technology*, `https://nvd.nist.gov/vuln/search/results?form_type=Advanced&cves=on&cpe_version=cpe%3a%2fa%3agnu%3alibtasn1%3a3.5`

[10] Rawat, S., Jain V., Kumar, A., Cojocar, L., Giuffrida, C., Bos, H. *VUzzer: Application-aware Evolutionary Fuzzing*, 2017.

[11] Savich, M. *avatar-panda*, `https://github.com/m3lixir-nyu/avatar-panda`

[12] Savich, M. *avatar-qemu*, `https://github.com/m3lixir-nyu/avatar-qemu`

[13] Savich, M. *avatar2*, `https://github.com/m3lixir-nyu/avatar2`

[14] Savich, M. *io_taint*, `https://github.com/m3lixir-nyu/io_taint`

[15] Savich, M. *tdff*, `https://github.com/m3lixir-nyu/tdff`

[16] Schwartz, E., Avgerinos, T., Brumley D. *All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)*, 2010.

[17] Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G. *Driller: Augmenting Fuzzing Through Selective Symbolic Execution*, 2016.

[18] *Technical "whitepaper" for afl-fuzz*, `https://lcamtuf.coredump.cx/afl/technical_details.txt`

[19] Yun, I., Lee, S., Xu, M., Jang, Y., Kim, T. *QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing*, 2018.