2024 International Wireless Communications and Mobile Computing (IWCMC) | 979-8-3503-6126-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/IWCMC61514.2024.10592348

Exploring the Frontiers of Firmware Fuzzing: μAFL's Application on Cortex M4 and Unix Programs

Safayat Bin Hakim
Department of Information Systems
University of Maryland, Baltimore County
Baltimore, MD 21250 USA
Email: shakim3@umbc.edu

Muhammad Adil
Computer Science and Engineering
The State University of New York at Buffalo
Buffalo, NY 14260 USA
Email: muhammad.adil@ieee.org

Jordi Mongay Batalla
Institute of Telecommunications
Electronics and Information Technology
Warsaw University of Technology
Warsaw, Poland
Email: jordi.mongay.batalla@pw.edu.pl

Constandinos X. Mavromoustakis
School of Sciences and Engineering
Department of Computer Science
University of Nicosia
Nicosia, Cyprus
Email: mavromoustakis.c@unic.ac.cy

Houbing Herbert Song
Department of Information Systems
University of Maryland, Baltimore County
Baltimore, MD 21250 USA
Email: h.song@ieee.org

***Abstract*—The aim of this study is to investigate into μAFL, a non-intrusive, feedback-driven fuzzing framework, evaluated on Cortex M4 embedded systems and Unix platforms, focusing on the STM32F407VE Cortex M4 microcontroller. By leveraging the SEGGER J-Trace Pro for trace collection, it demonstrates μAFL's utility beyond its traditional scope, showcasing its efficacy in both embedded and general-purpose computing environments. Our analysis, enriched by juxtaposing μAFL's capabilities with traditional AFL, emphasizes the adaptability and effectiveness of fuzzing methodologies in firmware security enhancement. Furthermore, the study provides a deep understanding of fuzzing execution on different hardware, presenting an execution strategy for the STM32F407VE that highlights the framework's potential in identifying vulnerabilities, evidenced by tests on specific firmware programs such as an LED blinking program integrated with semihosting breakpoints and ETM tracing. The use of uninitialized memory sections and strategically placed breakpoints offers significant insights into the firmware's execution flow. The results of our comparative analysis clearly show that μAFL excels at uncovering vulnerabilities, reinforcing the need for evolving fuzzing methodologies to build stronger security systems for embedded devices. This contribution underscores the importance of refining fuzzing techniques to meet the intricate security demands of contemporary computing environments.**

***Index Terms*—firmware fuzzing, μAFL, STM32F407VE, Embedded Trace Macrocell, firmware security**

I. Introduction

Firmware vulnerabilities present a considerable challenge to the security and operational integrity of embedded systems, potentially creating avenues for unauthorized access and control. Addressing these vulnerabilities is paramount, and fuzzing has emerged as a key technique in this endeavor. By applying invalid, unexpected, or random data inputs to programs and analyzing their responses, fuzzing helps identify weaknesses in software that could be exploited by attackers [1], [2]. While traditional fuzzing tools like American Fuzzy Lop (AFL) have demonstrated their efficacy in desktop and server environments, the unique constraints and requirements of embedded systems necessitate specialized approaches.

μAFL emerges as an innovative solution tailored to these challenges [3]. As a non-intrusive, feedback-driven fuzzing framework, μAFL leverages hardware features such as the Embedded Trace Macrocell (ETM) to conduct efficient, coverage-guided fuzzing on embedded firmware. This approach marks a promising advancement in enhancing the security of embedded systems. To date, explorations of μAFL's capabilities have been notably concentrated on platforms such as the NXP `TWR-K64F120M`. This focus lays the groundwork for further investigations into the framework's adaptability and efficacy across a wider range of microcontroller boards.

In light of this, our paper aims to extend the exploration of μAFL's utility to include a more diverse array of embedded hardware, with a particular emphasis on the `STM32F407VE` Cortex M4 microcontroller. Through a thorough examination of the hardware-software setup required for this microcontroller, including the utilization of its ETM and the J-Trace Pro debugger for efficient trace collection and firmware fuzzing, we provide an in-depth analysis of μAFL's applicability to a broader hardware context. Moreover, by comparing μAFL's performance with that of traditional AFL in the fuzzing of common Unix programs, we deliver insights into the comparative effectiveness of these tools across various computing environments.

Our work not only underscores μAFL's versatility and potential in enhancing the security of an expanded range of embedded systems but also supports reproducibility and further research in firmware security. This effort represents

979-8-3503-6126-1/24/$31.00 ©2024 IEEE

a significant stride forward in our collective endeavor to bolster the defenses of embedded systems against firmware vulnerabilities.

The remainder of this paper is organized as follows: Section II provides a background on fuzzing and the μAFL approach. Section III introduces the preliminaries, including overviews of AFL and the methodology behind μAFL. Section IV describes adapting μAFL for the `STM32F407VE` board. Section V outlines our experimental setup, followed by results and analysis in Sections VI and VII. We discuss our findings in Section VIII. Finally, Section IX concludes the paper, suggesting directions for future research.

II. Background

The landscape of software testing has been profoundly reshaped by the advent of fuzzing tools, among which American Fuzzy Lop (AFL) stands out for its general-purpose application [4]. AFL revolutionized the approach to uncovering vulnerabilities by instrumenting code to monitor execution paths, thus optimizing the search for unexplored code regions. However, the unique demands of embedded systems—characterized by constrained resources and specific operational requirements—necessitate a tailored approach [5]. Enter μAFL, a tool designed to bridge this gap by targeting firmware fuzzing within microcontrollers. Diverging from AFL's source code modification strategy, μAFL adopts a hardware-in-the-loop methodology, leveraging embedded system development tools and hardware features like ARM's Embedded Trace Macrocell (ETM) for non-intrusive feedback collection [6].

The transition to μAFL is exemplified in our choice of the `STM32F407VE` [7] board for our study. This board, equipped with an ARM Cortex M4 processor, mirrors the `TWR-K64F120M` [8] in its processor architecture, offering a comparable testing ground with advanced debugging features. Central to our investigation is the ETM unit of the `STM32F407VE`, providing real-time instruction tracing crucial for the nuanced analysis required in firmware fuzzing. This capability, coupled with the board's Data Watchpoint and Trace (DWT) unit for comprehensive data and event monitoring, underscores μAFL's potential in pinpointing vulnerabilities more effectively within embedded systems, thereby enhancing their security and reliability.

III. Preliminaries

A. AFL Overview

AFL stands as a cornerstone in the domain of fuzzing tools, renowned for its effective utilization of compile-time instrumentation to enhance software testing. By ingeniously transforming target binaries to monitor execution paths, AFL facilitates a feedback-driven fuzzing process that prioritizes unexplored code regions, thereby optimizing coverage and efficiency in vulnerability detection [9] [10]. Unlike μAFL, which employs hardware features for non-intrusive feedback collection, AFL's method involves inserting additional code into the program at compile time, using tools like `afl-gcc` or `afl-clang`. This process not only aids in identifying executed code segments but also in mapping out the application's behavior under various input conditions. Furthermore, AFL's unique approach to coverage mapping through a bitmap file exemplifies its capability to adaptively guide the fuzzing input generation towards areas less traversed. The comparison between AFL and μAFL underscores AFL's significance in laying the groundwork for feedback-driven fuzzing, highlighting its instrumental role in advancing the field of software security through meticulous exploration of potential vulnerabilities.

B. μAFL Approach

The μAFL framework introduces a novel, non-intrusive feedback-driven fuzzing methodology specifically tailored for microcontroller firmware, distinguishing itself from AFL by leveraging the Embedded Trace Macrocell (ETM) hardware feature [11]. This approach eliminates the need for conventional AFL instrumentation, thus bypassing software modification and the associated overhead. μAFL's utilization of ETM allows for comprehensive instruction trace collection directly from the hardware, offering detailed insights into branch execution without altering the firmware's operational integrity. Key to its operation is the integration of Linear Code Sequence And Jump (LCSAJ) analysis, which processes the ETM output efficiently, facilitating the identification of unexplored code paths with minimal performance impact [11]. Moreover, μAFL employs a combination of online trace collectors and offline trace analyzers to refine the collected data, filtering out irrelevant information and enhancing the overall efficiency of the fuzzing process. This strategic approach addresses the challenges associated with fuzzing stripped binaries in firmware, which are notably difficult to instrument, thereby extending the applicability of fuzzing techniques to a wider range of embedded systems.

IV. Adapting μAFL

A. Target Platform

Our research adapts the μAFL framework for use with the `STM32F407VE` Cortex M4 board, a choice driven by the board's robust support for critical features essential for effective fuzzing, such as the Embedded Trace Macrocell (ETM). This adaptation marks a shift from the initial implementations on platforms like the NXP `TWR-K64F120M`, primarily chosen for their ease of integration with μAFL's non-intrusive feedback-driven fuzzing methodology. The `STM32F407VE` was selected for its widespread availability, superior hardware capabilities, and compatibility with μAFL's requirements for detailed instruction trace collection through ETM, facilitating a comprehensive evaluation of the fuzzing process on embedded systems.

B. Software Configuration

The integration of μAFL into the `STM32F407VE`'s firmware development process involved several critical steps. Utilizing the KEIL MDK, we compiled the target firmware, ensuring the generated `.axf` file was optimally configured for μAFL's

fuzzing process. An `.axf` file is an executable file format containing compiled code, data, and debug information, generated by Arm toolchains like KEIL MDK or GCC, which can be directly loaded and executed on Arm-based microcontrollers. Special attention was paid to the ETM configuration to guarantee efficient trace collection, necessitating adjustments to μAFL's default settings to align with the `STM32F407VE`'s hardware specifications. This process highlighted the importance of a meticulous setup to enable effective trace data capture, which is paramount for the success of the fuzzing operation. Challenges encountered during the adaptation, such as compatibility issues between μAFL and the target platform's software environment, were systematically addressed through targeted modifications to μAFL's source code and the deployment of custom scripts designed to streamline the trace collection process.

These adjustments to μAFL, tailored to the `STM32F407VE` board, underscore the framework's flexibility and the potential for its application across a diverse range of embedded systems. Through this adaptation process, we aim to demonstrate μAFL's capabilities in uncovering vulnerabilities within the firmware, thereby contributing to the enhancement of software security in critical embedded applications.

V. Methodology

A. Environment Setup and Preliminary Configuration

In this section, we briefly discuss the key points regarding setting up the environment for our experiments. Our setup process involves configuring both hardware and software components to effectively use the μAFL framework for fuzzing the `STM32F407VE` Cortex M4 board firmware.

Fig. 1. Fuzzing Trace Reference board `STM32F407VE` firmware with J-Trace Pro debug dongle [12]

B. Setting up essential Environment Variables

The environment variables in Listing 1 are used before executing the `ETMFuzz`.

```
1  $ export AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES=1
2  $ export AFL_SKIP_CPUFREQ=1
```

Listing 1. Common environment variables before initiate fuzzing process

The `AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES=1` variable tells AFL to not stop fuzzing when a crash is missed. This can be useful in situations where it's expected that not all crashes will be caught due to some kind of limitation, like a time constraint or resource limit.

The `AFL_SKIP_CPUFREQ=1` variable tells AFL to skip the checks it usually does for CPU frequency scaling. In normal operations, AFL checks to ensure the CPU is running at full speed to ensure maximum efficiency for the fuzzing process, but setting this variable allows the fuzzing to occur regardless of the CPU's speed.

1) Changing Configuration Files for J-Trace PRO and `STM32F407VE`: To set up the μAFL environment, we first made adjustments in the `device.h` file, specifically changing the `JLINK_SERIAL_NO` and `DEVICE_NAME` to match our hardware setup. This customization is crucial for the J-Trace device to correctly interface with our chosen microcontroller board, facilitating precise trace collection during fuzzing sessions.

2) Tracing Configuration in `trace.h` File: We tailored the `trace.h` file from the official μAFL GitHub repository [13] to fit our target microcontroller, `STM32F407VE`. This involved disabling configurations for other devices and ensuring the trace functionality is properly set up for our experiments.

3) Additional Steps to Fuzz with `STM32F407VE`: Integrating μAFL into the firmware's build process and preparing the microcontroller for fuzzing encompassed compiling the firmware with KEIL MDK and using the J-Trace Pro for trace data collection. These steps, outlined in detail, are foundational to our methodology, enabling us to conduct thorough fuzzing tests on the `STM32F407VE` applications.

```
1  $ arm-none-eabi-objcopy -O binary ./led-blinking.axf ./
       STM32F407.bin
2  $ ../ETMFuzz_Src/ETMFuzz -t 50 -i in_hello_world -o
       out_STM32F407_led ./STM32F407.bin
```

Listing 2. Commands used to fuzz `STM32F407VE` applications

This detailed setup description not only elucidates the experimental groundwork but also ensures reproducibility and transparency of our research methodology.

4) Command-Line Operations for Fuzzing: Following the configuration adjustments, we proceed with the firmware preparation and fuzzing initiation using specific command-line operations. Here, we use the `arm-none-eabi-objcopy` utility to convert the input file `led-blinking.axf` into a binary file format (`-O binary`). The output file is named `STM32F407.bin` and is placed in the current directory (`./`).

To initiate fuzzing with μAFL, we employ the following command sequence:

- `-t 50` sets the timeout to 50 seconds.
- `-i in_hello_world` designates `in_hello_world` as the input directory.
- `-o out_STM32F407_led` specifies `out_STM32F407_led` as the output directory.

- `./STM32F407.bin` identifies the binary file to be fuzzed.

This command sequence is crucial for conducting coverage-guided fuzz testing on the `STM32F407.bin` firmware using the μAFL tool, demonstrating our methodology's practical application.

C. Code Availability

Our GitHub repository is based on foundational fuzzing work [13], adapted for the `STM32F407VE` Cortex M4 board. It includes source code, environment setup configurations, and a README detailing setup and execution instructions, facilitating reproducibility and further exploration. Repository link: https://github.com/sbhakim/uAFL-STM32F407VE-Applications.git.

VI. Experimental Setup

A. Preparation of Test Cases

To initiate the fuzzing process, we created a dedicated folder within the `microAFL/custom_eval` directory to house our test evaluations. This involved the generation of several `.axf` files, including `helloworld-fibonacci.axf`, `helloworld_hardfault.axf`, and `led-blinking.axf`, utilizing the KEIL Microcontroller Development Kit (MDK) on a Windows platform. The `.axf` files serve as executable formats containing compiled code, data, and debug information, ready to be loaded onto the `STM32F407VE` board for fuzzing with μAFL.

B. KEIL MDK Configuration and Debugging

We configured the KEIL MDK to select the appropriate debug dongle (J-Link / J-Trace) and to enable ETM tracing, crucial for capturing detailed execution traces during fuzzing with μAFL. Specific attention was paid to the debug and trace settings to optimize the tracing capability and ensure efficient real-time monitoring. This setup phase was critical for enabling the high-fidelity tracing required for effective fuzz testing of the `STM32F407VE` board.

C. Firmware Compilation and Transfer

The firmware, designed to include various peripherals such as GPIO and UART for the `STM32F407VE` application, was compiled to generate the `.axf` file. Despite encountering compilation warnings, they were deemed non-critical and thus not addressed, focusing on functionalities relevant to the fuzzing objectives with μAFL. The compiled `.axf` file was then transferred to a Linux environment, where it was positioned for fuzzing within the `microAFL/custom_eval` directory, marking the final preparation step before initiating the fuzzing process with μAFL.

VII. Results and Analysis

A. Fuzzing a simple bug-prone C program with `afl-gcc`

Listing 3 is a simple file reader that reads structures of type calculator from a file [14]. It performs calculations based on the data read and performs various operations using dynamic memory allocation. The `vulfoo` function is called with a filename as a parameter, which is passed as a command-line argument.

Here are the bugs present in the code:

Double Free: After `buff1` is freed, it is freed again if `size1/2` equals 0. This is a double free vulnerability because it's undefined behavior to free a pointer that has already been freed.

Use After Free: If `size1/3` equals 0, the program tries to access buff1 after it has been freed. This is a use-after-free vulnerability, as the program is accessing memory that it no longer owns.

Division by Zero: The program calculates `size2` as the division of `calc.num1` by `calc.num2`. If `calc.num2` is 0, this will result in a division by zero error, which is undefined behavior in C.

Buffer Overflow: The program uses `memcpy` to copy the contents of `calc.data` into `buff1` and `buff2`. However, the size of `calc.data` is 10 bytes, while `buff1` and `buff2` are allocated `size1` and `size2` bytes respectively. If `size1` or `size2` are less than 10, this results in a buffer overflow as `memcpy` tries to copy more data than the destination buffer can hold.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct calculator{
    int num1;
    int num2;
    char data[10];
};

int vulfoo (char *filename){
FILE *fp;
struct calculator calc;

fp = fopen(filename, "r");

if (fp == NULL){
    printf ("\n Can't open file or file doesn't exist.
         \n");
    exit(0);
}

while(fread(&calc, sizeof(calc), 1, fp) > 0){
    int size1 = calc.num1 + calc.num2;
    char* buff1 = (char*) malloc(size1);

    memcpy(buff1, calc.data, sizeof(calc.data));
    free(buff1);
    /* double free error */
    if (size1/2==0){
        free(buff1);
    } else if (size1/3 == 0){
        /* Use after free vulnerability */
        buff1[0] = 'a';
}

int size2 = calc.num1 / calc.num2;
char* buff2 = (char*) malloc(size2);
memcpy(buff2, calc.data, sizeof(calc.data));
free(buff2);
}

fclose(fp);
return 0;
}

int main(int argc, char **argv){
    vulfoo(argv[1]);
}
```

Listing 3. Demonstration of Common Vulnerabilities in `vulfoo.c` program: Double Free, Use After Free, Division by Zero, and Buffer Overflow

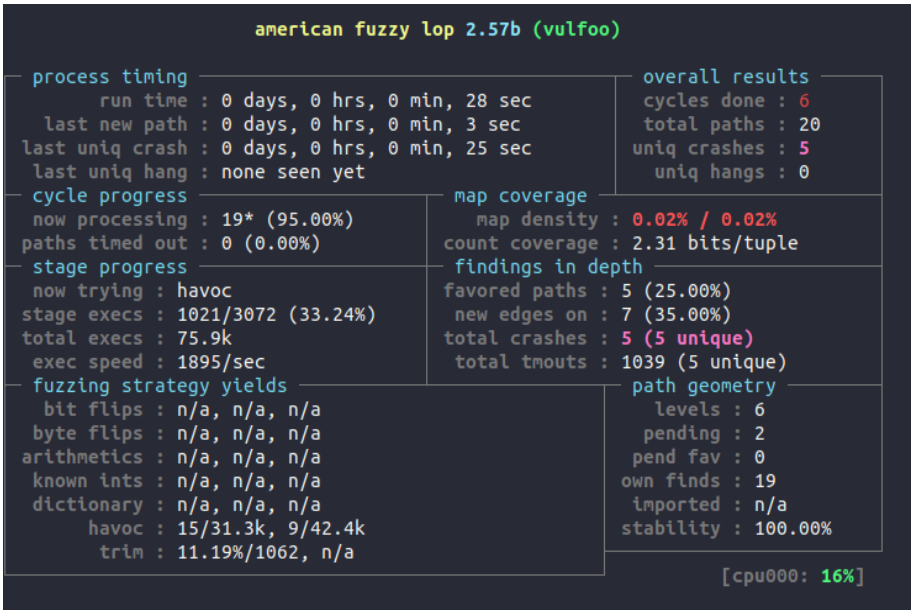

Fig. 2. Printed status of AFL fuzzing of `vulfoo` program

B. Interpreting the Fuzzing output

Fig. 2 depicts the output of the fuzzing process of Listing 1. In the following, we discuss the most important aspects of the result.

Total Execs: This refers to the total number of times the program being fuzzed was executed. In your output, this is 75.9k (or 75,900 times). Each execution tests a slightly different input to the program, with the goal of finding inputs that cause the program to behave unexpectedly.

New Paths: The process of fuzzing involves an exploration of distinct "paths" through the subject program's codebase [15]. Each of these paths corresponds to a unique sequence of instructions executed during runtime. The identification of a "new path" denotes an execution instance that successfully reached a previously unexplored segment of the program's code. As per the provided output data, the most recent path discovery occurred merely three seconds prior, resulting in a cumulative total of 20 distinct paths discovered thus far in the process.

Crashes: If the program crashes during an execution, this represents a potential vulnerability. The output shows both the total number of crashes and the number of unique crashes, with unique crashes representing different, non-duplicative potential vulnerabilities. In your output, there are 5 total crashes, all of which are unique.

Map Density: This measures the percentage of the program's code that has been reached by the fuzzer's testing [16]. Here, the map density is quite low at 0.02%, indicating that the program being fuzzed has a large codebase, or that the fuzzer has so far only been able to reach a small portion of the code.

Uniq crashes / Uniq hangs: The terms "uniq crashes" and "uniq hangs" in the AFL output refer to the distinct instances of crashes and hangs that were encountered during the fuzzing process. It's important to note that these "unique" instances do not necessarily correspond to entirely different types of vulnerabilities or bugs [17]. For instance, two crashes resulting from similar "divide by zero" bugs would be considered unique, even though the underlying bug is of the same type. This is because each occurrence is distinct and can independently cause the program to crash. Therefore, while AFL provides valuable insights into potential issues within a codebase, it does not provide specific information about the type or category of the underlying vulnerabilities, such as buffer overflow.

Favored Paths: This refers to the number of paths that are currently deemed "interesting" by AFL and are prioritized for further exploration. These paths have shown to result in new coverage and potentially new findings. The number of favored paths can provide insights into how effectively the fuzzer is in discovering new parts of the program to explore. In your output, 5 out of the 20 total paths (25%) are favored, which suggests that these paths have shown to be especially fruitful in terms of achieving new coverage or findings.

C. Fuzz Bitmap

AFL uses bitmaps to manage basic block transitions within the tested program [18] [9], forming both local and global bitmaps. The local bitmap corresponds to each input file, monitoring specific code coverage to assess its worth for further examination. The fuzzing manager amalgamates all local bitmaps into a global bitmap, guiding the mutation-based fuzzing process towards untouched code segments and identifying novel paths [19]. This information can be found in the `fuzz_bitmap` file within the "`output`" directory. In this file, each line (Listing 5) represents 16 bytes of data with

asterisks indicating recurring content. The bitmap is structured in little-endian format, with each byte signifying a branch in the program.

```
$ hexdump fuzz_bitmap
0000000 ffff ffff ffff ffff ffff ffff ffff ffff
*
0002f90 ffff ffff ffff ffff feff ffff ffff ffff
0002fa0 ffff ffff ffff ffff ffff ffff ffff ffff
*
0003240 ffff ffff ffff ffff ffff ffff ffff fff0
0003250 ffff ffff ffff ffff ffff ffff ffff ffff
*
00033c0 ffff ffff ffff ffff feff ffff ffff ffff
00033d0 ffff ffff ffff ffff ffff ffff ffff ffff
*
0004090 feff ffff ffff ffff ffff ffff ffff ffff
00040a0 ffff ffff ffff ffff ffff ffff ffff ffff
*
0004110 ffff ffff ffff fffe ffff ffff ffff ffff
0004120 ffff ffff ffff ffff ffff ffff ffff ffff
*
0004c40 ffff ffc0 ffff ffff ffff ffff ffff ffff
0004c50 ffff ffff ffff ffff ffff ffff ffff ffff
```

Listing 4. Fuzz Bitmap contains coverage map

AFL doesn't map branches to bitmap bytes directly, but assigns random two-byte constant IDs to each branch. The XOR operation is used to combine the IDs of the current and last observed branch, capturing both the current branch and the unique path leading to it. A hashing function then determines the bitmap entry representing the branch combination, and each exercise of a specific branch combination increments the corresponding byte in the bitmap. However, while the bitmap size is 64KB, the output displays non-repeated sections, and variations such as `fffe`, `feff`, and `ffff` in the bitmap values represent different hit counts for program edges.

D. Fuzzing a Vulnerability-Prone STM32F407VE Program with μAFL

In our research, we examine the use of μAFL for testing on the `STM32F407VE` microcontroller, utilizing the J-Trace PRO debugger. Detailed information and all required files for this experiment can be found in the `/microAFL/custom_eval` directory, ensuring the experiment can be easily replicated.

The experiment centers on the fuzzing of an LED blinking program to assess μAFL's capability in uncovering embedded firmware vulnerabilities. Utilizing the `arm-none-eabi-objcopy` utility, the `./STM32F407.bin` binary file, derived from `./led-blinking.axf`, is prepared as the subject for fuzzing. Configured to enforce a strict timeout of 5000 milliseconds for each test case, μAFL initiates the fuzzing process, analyzing inputs from the `in_hello_world` directory and compiling results into `out_STM32F407_led_blinking`. This procedural approach underscores the adaptability and precision of μAFL in identifying potential security vulnerabilities within the targeted embedded system.

VIII. Discussion

A. Coverage mapping, tracking and optimizing through AFL Instrumentation

AFL employs a mechanism known as instrumentation to transform a target binary, enabling it to provide feedback regarding which parts of the code are exercised during fuzzing. This mechanism is executed at compile-time, where the AFL compiler wrappers like `afl-gcc` or `afl-clang` are used to inject additional code into the program. The injected code essentially forms a lightweight coverage tracking mechanism which helps AFL to understand and assess the code paths traversed. The output message "Instrumented 8 locations (64-bit, non-hardened mode, ratio 100%)" indicates that `afl-gcc` has successfully instrumented the provided source file, `vulfoo.c`, modifying eight distinct code locations to aid in tracking the code execution path during fuzzing.

```

$ afl-gcc vulfoo.c -o vulfoo
afl-cc 2.57b by <lcamtuf@google.com>
afl-as 2.57b by <lcamtuf@google.com>
[+] Instrumented 8 locations (64-bit, non-hardened mode,
    ratio 100%).
```

Listing 5. Compile-time instrumentation by `afl-gcc`

Each `__afl_` prefixed entity in Listing 6, like `__afl_area_ptr` or `__afl_fork_pid`, corresponds to a unique function or variable inserted into the program to monitor its execution, enhancing AFL's ability to effectively perform fuzzing operations. The symbols like `0000000000004018`, `b`, `t`, `B` etc., refer to the compile-time memory address offset and type of symbol respectively, where `t` denotes a local text/code segment, `b` is for local uninitialized data section (BSS), and `B` is for global uninitialized data section.

```

$ nm vulfoo | grep afl_
0000000000004018 b __afl_area_ptr
000000000000192e t __afl_die
0000000000004028 b __afl_fork_pid
0000000000001849 t __afl_fork_resume
000000000000178b t __afl_forkserver
00000000000017b1 t __afl_fork_wait_loop
0000000000004038 B __afl_global_area_ptr
0000000000001620 t __afl_maybe_log
0000000000004020 b __afl_prev_loc
0000000000001648 t __afl_return
0000000000001650 t __afl_setup
0000000000001936 t __afl_setup_abort
0000000000004030 b __afl_setup_failure
0000000000001671 t __afl_setup_first
0000000000001630 t __afl_store
000000000000402c b __afl_temp
```

Listing 6. AFL instrumented in various places of `vulfoo` program

B. μAFL leverages ETM for Instruction Trace Collection over traditional instrumentation

The document introduces a novel approach, μAFL, which utilizes the ETM hardware feature to generate an instruction trace, eliminating the need for AFL instrumentation. This approach is non-intrusive to the firmware, requiring no software instrumentation and introducing no additional overhead. It addresses the challenge of dealing with stripped binaries in firmware, which can be difficult to instrument. ETM by default collects all branch information, providing a comprehensive instruction trace for a testcase. Additionally, μAFL employs online trace collectors and offline trace analyzers to filter irrelevant ETM packets, thereby enhancing performance.

C. Semhosting and BKPT instructions in μAFL

Semihosting is a feature available on ARM Cortex microcontrollers, which allows embedded programs to leverage the capabilities of an attached computer during the operation of a debugger [20] [21], a feature that proves beneficial in μAFL. A key aspect of this process is the halting of the CPU target by a debugger agent, achieved through the use of a breakpoint instruction, such as `BKPT 0xEF` or `BKPT 0xFF` which are used in μAFL. This halting signals to the host that the target is requesting a semihosting operation, which is executed by the host while the CPU remains halted, with the result returned before the processor continues with its program.

```
#if !defined (__SEMIHOST_HARDFAULT_DISABLE)

__attribute__((naked))
void HardFault_Handler(void){
    __asm("bkpt 0x11\n\t");
    __asm(  ".syntax unified\n"
    "MOVS   R0, #4  \n"
    "MOV    R1, LR  \n"
    "TST    R0, R1  \n"
    "BEQ    _MSP    \n"
    "MRS    R0, PSP \n"
    "B  _process      \n"
    "_MSP:  \n"
    "MRS    R0, MSP \n"
    "_process:      \n"
    "LDR    R1,[R0,#24] \n"
    "LDRH    R2,[r1] \n"
    "LDR    R3,=0xBEAB \n"
    "CMP     R2,R3 \n"
    "BEQ    _semihost_return \n"
    "B . \n"
    "_semihost_return: \n"
    "ADDS    R1,#2 \n"
    "STR    R1,[R0,#24] \n"
    "MOVS   R1,#32 \n"
    "STR R1,[ R0,#0 ] \n"
    "BX LR \n"
    ".syntax divided\n") ;
}
#endif
```

Listing 7. Custom HardFault Handler for Semihosting Operations without Debugger

The code in Listing 7 [13] exhibits a handling mechanism for semihosting operations in situations where the debugger is disconnected. Semihosting operations like `printf` calls can cause an application to hang if the debugger is not present, triggered by the "`BKPT 0xAB`" instruction which, in the absence of a debugger, incites a hard fault. The default handler for such a fault often results in an infinite loop, causing the application to freeze. The code in Listing 3, however, implements a custom hard fault handler which inspects the instruction that induced the hard fault; if the culprit is "`BKPT 0xAB`", it returns control back to the user application, allowing the application incorporating semihosting operations to function to some degree without a connected debugger. The code begins with an inline assembly '`bkpt`' instruction, which triggers a breakpoint exception for debugging purposes. Rest of the code handles HardFault exceptions by performing a stack frame analysis, checking for a specific identifier in the faulting instruction's address, and if found, modifying it to bypass the fault, thereby intercepting semihosting calls and preventing HardFaults, before resuming program execution.

In the original μAFL project [22], the code segment for the `hello_world` project is located in the `microAFL/microAFL_eval/twrk64f120m_hello_world/` directory. This project was originally implemented for the NXP `TWR-K64F120M` board by the authors of μAFL. However, in our adaptation for the `STM32F407VE` board, no errors were detected upon its inclusion in the KEIL project, resulting in successful generation of the `.axf` file.

IX. Conclusion

This study has illustrated the pivotal role of μAFL in advancing the security measures for embedded systems, particularly through the lens of the `STM32F407VE` Cortex M4 microcontroller. The integration of the Embedded Trace Macrocell (ETM) and SEGGER J-Trace Pro debugger within the μAFL framework has not only demonstrated a non-intrusive method for vulnerability identification but also emphasized the tool's adaptability to diverse hardware environments. The juxtaposition of μAFL's performance against that of traditional AFL, especially in the fuzzing of Unix programs, sheds light on the unique challenges and prospects of applying fuzzing tools across various computing platforms. Our research underlines the efficacy of μAFL in identifying firmware vulnerabilities, marking a significant contribution towards the fortification of embedded systems against potential threats. Future endeavors in this domain should continue to explore and refine fuzzing methodologies, ensuring that security frameworks remain resilient in the face of evolving cyber threats.

Acknowledgment

This research was partially supported by the U.S. National Science Foundation through Grant No. 2317117 and Grant No. 2309760. The authors would like to thank the Department of Computer Science and Engineering at the University at Buffalo for their continuous support and provision of resources. Special thanks go to Dr. Ziming Zhao for his valuable guidance and for providing access to the SEGGER J-Trace board, which was instrumental in conducting the experiments presented in this paper.

References

[1] J. Liang, Y. Jiang, M. Wang, X. Jiao, Y. Chen, H. Song, and K.-K. R. Choo, "Deepfuzzer: Accelerated deep greybox fuzzing," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 6, pp. 2675–2688, 2021.

[2] Z. Yu, H. Wang, D. Wang, Z. Li, and H. Song, "Cgfuzzer: A fuzzing approach based on coverage-guided generative adversarial networks for industrial iot protocols," *IEEE Internet of Things Journal*, vol. 9, no. 21, pp. 21 607–21 619, 2022.

[3] W. Li, J. Shi, F. Li, J. Lin, W. Wang, and L. Guan, "μafl: non-intrusive feedback-driven fuzzing for microcontroller firmware," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1–12.

[4] Google, "American fuzzy lop," Access year, accessed: Insert access date here. [Online]. Available: https://github.com/google/AFL

[5] 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.

[6] A. Vergé, N. Ezzati-Jivan, and M. R. Dagenais, "Hardware-assisted software event tracing," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 10, p. e4069, 2017.

[7] *STM32F405xx STM32F407xx Datasheet*, STMicroelectronics, Accessed: 2024, accessed on Feb 12, 2024. [Online]. Available: https://www.st.com/resource/en/datasheet/stm32f407vg.pdf
[8] "Freescale mqx™ rtos 4.1.0 twr-k64f120m release notes," NXP Semiconductors, Tech. Rep., Accessed: 2024, accessed on Feb 12, 2024. [Online]. Available: https://www.nxp.com/docs/en/release-note/MQXTWRK64RN.pdf
[9] A. Fioraldi, A. Mantovani, D. Maier, and D. Balzarotti, "Dissecting american fuzzy lop–a fuzzbench evaluation-rcr report," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 2, pp. 1–4, 2023.
[10] M. Zalewski, "American fuzzy lop (2.52b)," https://lcamtuf.coredump.cx/afl/, 2024, accessed on: Feb 13, 2024.
[11] W. Li, "[ICSE 2022] µAFL: Non-intrusive Feedback-driven Fuzzing for Microcontroller Firmware," https://www.youtube.com/watch?v=exampleURL, 2022, [Online; accessed 14-February-2024].
[12] "Um08001 j-link / j-trace user guide," https://wiki.segger.com/UM08001_J-Link_/_J-Trace_User_Guide, 2024, accessed on: Feb 14, 2024.
[13] MCUSec, "uAFL: Non-intrusive Feedback-driven Fuzzing for Microcontroller Firmware," https://github.com/MCUSec/microAFL, 2024, accessed on Aug 15, 2024.
[14] H. Shah, "[Fuzzing with AFL] How to Fuzz a Simple C Program with AFL on Linux," https://www.youtube.com/watch?v=NiGC1jxFx78, 2020, accessed: 2024-02-12.
[15] *AFL User Guide*, 2024, accessed on Feb 12, 2024. [Online]. Available: https://afl-1.readthedocs.io/en/latest/user_guide.html
[16] "Afl internals - stats, counters and the ui," https://www.core.gen.tr/posts/007-afl-stats-counters-and-ui/, 2024, accessed on Feb 12, 2024.
[17] M. Macnair and M. Nedyak, "Fuzzing with afl workshop," https://github.com/mykter/afl-training, 2023, accessed: 2023-06-06.
[18] "Fuzzing part 2: Fuzzing with afl," https://sayfer.io/blog/fuzzing-part-2-fuzzing-with-afl/, accessed: 2023-06-05.
[19] S. Mallissery and Y.-S. Wu, "Demystify the fuzzing methods: A comprehensive survey," *ACM Computing Surveys*, vol. 56, no. 3, pp. 1–38, 2023.
[20] A. E. Messaoudi. (2021) Introduction to arm semihosting. Accessed: 2023-06-05. [Online]. Available: https://interrupt.memfault.com/blog/arm-semihosting
[21] "ARM Semihosting - native but slow Debugging," https://www.codeinsideout.com/blog/stm32/semihosting/, accessed: 2023-06-05.
[22] MCUSec, "microafl: Non-intrusive feedback-driven fuzzing for microcontroller firmware," 2023, accessed: 2023-06-05. [Online]. Available: https://github.com/MCUSec/microAFL

APPENDIX

SUPPLEMENTARY MATERIAL

This section provides additional technical details and code examples that supplement the main content of our paper, particularly focusing on the practical application of μAFL for fuzzing a simple program on the `STM32F407VE` board.

A. Practical Application of μAFL on `STM32F407VE`

To demonstrate the adaptability of μAFL to embedded systems, we conducted an experiment with a simple LED blinking program on the `STM32F407VE` board. This example showcases the integration of essential configuration steps in KEIL MDK and the utilization of Embedded Trace Macrocell (ETM) for non-intrusive instruction tracing.

1) Configuration and Code Setup: The KEIL MDK was utilized for compiling and setting up the necessary files for the `STM32F407VE` board. The program, as depicted below, includes semihosting breakpoints and ETM tracing instructions to facilitate detailed analysis during the fuzzing process.

```c
#include "stm32f4xx.h"
#include "stm32f4xx_hal_gpio.h"
#include "stm32f4xx_hal_rcc.h"
#include <string.h>

void SystemClock_Config(void);


/* ETM tracing necessary for uAFL */
unsigned int etm_tc_len
     __attribute__((section(".non_init")));
unsigned char etm_tc[2000]
     __attribute__((section(".non_init")));
unsigned int etm_tc_idx = 0;
unsigned int etm_exit = 0;

void SystemClock_Config(void) {}

void delay(volatile int d) {
while (d--) {
 __NOP();
}}

int main(void) {
/* Configure the system clock */
SystemClock_Config();

/* Breakpoint before LED Initialization */
__asm("bkpt 0xEF\n\t");

/* Initialize GPIO for LED D2 (PA6) and LED D3 (PA7) */
GPIO_InitTypeDef GPIO_InitStruct;
__HAL_RCC_GPIOA_CLK_ENABLE();
GPIO_InitStruct.Pin = GPIO_PIN_8 | GPIO_PIN_9;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

/* Initialize the pins to be off (assuming LEDs are
     active high) */
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_8 | GPIO_PIN_9,
     GPIO_PIN_RESET);

/* Breakpoint before LED blinking loop */
while (1) {
/* Breakpoint before LED toggle */
HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_8);
HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_9);
delay(1000000 / 6); // Blink 3 times per second
/* Breakpoint after LED toggle */
}

/* Breakpoint after LED blinking loop */
__asm("bkpt 0xFF\n\t");

return 0;
}
```

Listing 8. Simple `STM32F407VE` LED blinking program integrated with semihosting BKPT instructions and ETM tracing

2) Insights from the Experiment: The inclusion of the Hardware Abstraction Layer (HAL) libraries abstracts the low-level hardware interactions, simplifying the firmware development for fuzzing. Breakpoints (`BKPT`) strategically placed at various stages of the program aid in the ETM's data collection process, providing valuable insights into the execution flow and potential vulnerabilities.

The use of uninitialized memory sections, designated by `__attribute__((section(".non_init")))` compiler directive, enables us to reserve space for dynamically generated test cases by μAFL, illustrating an innovative approach to embedded system fuzzing.

3) Observations and Debugging: Throughout the debugging process in KEIL MDK, the program's execution aligned with our expectations, demonstrating the effectiveness of the setup for instruction tracing and the potential of μAFL in identifying vulnerabilities within embedded systems.