Almiraj: A Method for Fuzzing Embedded Systems Leveraging Unicorn & AFL

Zach Heller
M.S. in Computer Science
Washington University in St. Louis
zheller@wustl.edu

Bryan Orabutt
PhD Student in Computer Engineering
Washington University in St. Louis
borabutt@wustl.edu

***Abstract*—Embedded systems are hard to fuzz. Cross-platform fuzzing (e.g. using a desktop to send fuzzing inputs to an embedded device) has large performance problems. Hardware limitations in communication slow fuzzing considerably, and monitoring state information is a task unto itself due to overhead and problems with fault detection. Crashes make the system lock up which halts communication from the embedded device. If one can get to the point of reliably observing crashes and communicating, resetting the state of an embedded device will usually require a reboot, which is a massive time sink. Additionally, the various architectures and OSes reduce the portability of an embedded fuzzer. Our goal was to create a portable method for fuzzing embedded systems. We propose Almiraj as a new method of fuzzing for vulnerable embedded code. By leveraging the architecture emulation of the Unicorn Engine, we bypass many hardware limitations and performance pitfalls of previously explored hardware fuzzers. We chose to target FreeRTOS running on a BeagleBone Black (BBB) as a proof of concept because of FreeRTOS's popularity and ubiquity as an embedded operating system. Any device that can be emulated by Unicorn should be able to be tested, allowing for a wide range of devices and their binaries to be fuzzed. We call our framework Almiraj after the mythical horned rabbit; our version is an American Fuzzy Lop with a unicorn horn.**

***Index Terms*—fuzzing, FreeRTOS, emulation, Unicorn, AFL**

I. Methodology

The original goal was to create a system for fuzzing embedded devices by using Unicorn and afl-unicorn to allow for fast fuzzing of emulated binaries. We could then "replay" emulated crashing inputs on target hardware to verify the errors were in the binary and not in emulation. Because the emulated code might need access to memory mapped IO at some point during execution, we were going to write an IO forwarding mechanism to allow IO access to Unicorn through JTAG (see Figure 1). This would allow Unicorn to read/write to memory mapped IO, and would also allow for replaying of inputs on the target device once a crash was detected. Unfortunately, we were overly ambitious and only got each component working individually to varying degrees.

A. Source code & reproduction of experiments

As of right now most of our source code works independently, but we have no software to integrate all parts successfully. We have working IO forwarding/JTAG code, working test harnesses for fuzzing, and somewhat working emulation code for Unicorn. We evaluated each piece independently of each other. All source code can be found in our GitHub repository located at https://github.com/BryanOrabutt/almiraj.

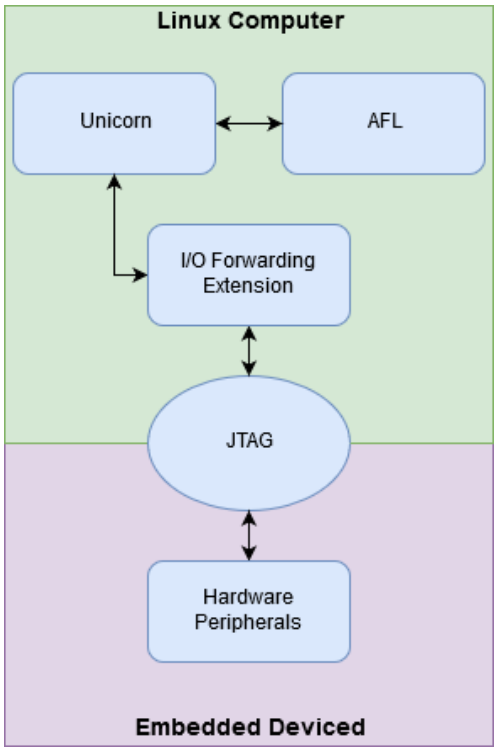

Fig. 1: The proposed fuzzing system we dubbed "Almiraj"

1) IO Forwarding/JTAG: Our IO forwarding mechanism comes in the form of a Python 2 library. We had to use Python 2 since Unicorn and afl-unicorn are not supported in Python 3. The file **src/openocd_scripts/OpenOCD.py** defines this library, and also acts as an executable to allow standalone testing of the library functions. The library implements all of the functionality that is needed to handle IO forwarding as well as some useful functionality for debugging and performance analysis.

In order to replicate the experiment we talk about in *Evaluation*, first build the code described in the *Emulation* subsection for the BBB. This is done easily using the makefile in the BBB directory. Place the resulting application binary (called **app**) on an SD card, along with the bootloader (called

MLO) from the **BBB/doc** folder. Place the SD card in the BBB and hold the boot button and apply power. The BBB will boot from the SD card and load the application binary into RAM at address 0x80000000.

Next you must build the TI fork of OpenOCD. The source code is located in **openocd/openocd/**. In this directory execute the following:

- ./bootstrap
- ./configure –enable-maintainer-mode –enable-ftdi –enable-xds110
- make
- sudo make install

Now, with the program running on the BBB connect the XDS110 JTAG emulator to the PC and the BBB. From **src/openocd_scripts/** folder, execute the **runit** script to start OpenOCD targeting the debug access port (DAP) on the BBB's ARM core. This script may need to be run as sudo/root. Once OpenOCD is running, execute the Python library and you will be presented with a prompt to enter commands. Halt the processor with the **halt** command and overwrite the string at address 0x80003000 using the **memwrite [address] [count] [size]** command. The arguments for this command are the address to begin writing to, the number of data elements to write, and the size of each element (e.g. w = word, h = half-word, b = byte). Make sure to include a NULL terminator in your string. Once the string is overwritten, resume execution.

With the modified program executing, attach a USB→Serial converter cable to the UART port on the BBB. Use a terminal emulator program such as Cutecom or screen, configured for 115200 baud, and view the messages being printed over the UART. You will see the string message you wrote into RAM being printed.

2) Emulation: To begin emulating you must install Unicorn. The easiest way to do this is to simply install afl-unicorn using the install scripts. This will cover installing Unicorn and afl-unicorn in one fell swoop. To do this execute the following from the **afl-unicorn/** directory:

- make
- sudo make install
- cd unicorn_mode
- sudo ./build_unicorn_support.sh

We have a somewhat working emulation script for Unicorn located in **unicorn_stuff/** called **freertos_emu.py**. This script emulates a FreeRTOS binary in Unicorn and adds hook functions in order to display executed instructions and register values. Unfortunately the emulation will eventually crash when the program counter reaches 0x80000920; a problem we are still trying to figure out.

Running this code is fairly easy. First a memory map describing the address range of each function is needed, described in JSON format. This can be created using the file **src/helper_scripts/disasm_parser.py** and providing an argument in the form of a filename of a disassembly text file. Then execute it in Python 2 and a stream of messages will be printed describing the emulation state. It can be useful to redirect this output to a file to easier viewing. The instructions and register values can be cross referenced to a disassembly dump of the target binary using **objdump** if the appropriate ARM toolchains are installed. To test other binaries, simply change the filename in the emulation script from target.bin to the filename of the binary you wish to emulate. The simple bare metal program used to test OpenOCD.py is included in this directory, called **app**. This file, however, does not emulate properly.

3) Fuzzing: afl-unicorn is a project Nathan Voss developed at Battelle, which he open-sourced and explains in his articles on Hacker Noon[1], [2]. His specialized fuzzer functions the same as AFL, but with a new mode to target binaries emulated by Unicorn. We successfully compiled afl-unicorn on a couple Linux distributions[1]. We verified our install of afl-unicorn was working with a sample test harness, sample target binary, and sample input binaries. This can be found in **almiraj_src/afl-unicorn/unicorn_mode/samples/simple/**. From there, execute **afl-fuzz -U -m none -i inputs -o output −− python2 test_harness.py @@** to start fuzzing in Unicorn mode. To fuzz a portion of our FreeRTOS binary (as far as our emulation will execute successfully), run the **run_fuzz** script from within the **src/fuzz_testing** directory. We explored many permutations of inputs and modified the test harness extensively, but we could not get the Unicorn engine to emulate correctly. Due to our struggle with emulation, we did not make a lot of progress in terms of fuzzing and thus cannot evaluate our fuzzing speed or efficacy.

II. Evaluation

We first tried to get all of the components of our project working independent of each other. We wrote a Python library to implement IO forwarding in JTAG through OpenOCD. We tried emulating target binaries in Unicorn without any fuzzing. Finally, we made an attempt to bring all of the pieces together with a simple UART bare metal program, but were not met with success.

A. IO Forwarding through JTAG

One of the most important parts of our project is having the ability to forward data over JTAG to real hardware peripherals. Because Unicorn only emulates the CPU and no hardware peripherals, we had to find a way to allow Unicorn to detect whether an address was physical memory, or memory mapped IO. In addition, we needed a mechanism that would allow IO access through JTAG if Unicorn needed to read or write to a memory mapped IO device.

The first approach that we took to doing this was to modify Unicorn's **uc_mem_read()** and **uc_mem_write()** functions directly. We added an extra parameter to specify a filename to each of the functions. The file was assumed to be of JSON format and the address passed into the function would be compared with the address range defined in the JSON. If a hit was found, we would know that the address was an IO device.

[1]The build scripts work on Ubuntu but require modification for other distributions.

This worked well, but we ran into roadblocks when trying to add JTAG libraries to implement the IO forwarding, leading to many build issues. Eventually this idea was abandoned in favor of using a simpler external interface, in this case we chose OpenOCD.

OpenOCD provides two mechanisms for communication with external software: a tcl remote procedure call (RPC) server, and a telnet server. We first tried writing a Python program that would allow us to use the RPC server. This would have given us more flexibility as there are some commands only available via the RPC server, and it also allows tcl scripts (containing OpenOCD commands) to be executed which would give us more functionality. Due to our unfamiliarity with tcl and the lack of documentation for using the RPC server, we eventually gave up this route in favor of using the OpenOCD telnet server which was a much simpler interface.

Our final approach worked very well. Using our Python library we were able to send commands to OpenOCD which allowed us to perform register operations and memory operations on the target device (BBB). To test and make sure it was working we put a simple bare metal UART program on the BBB and used our Python library to write a custom string to the address of the buffer that our UART was printing. We then resumed the execution of the processor and observed our serial terminal program and were able to see this custom string being printed.

B. Emulation in Unicorn

The Unicorn Engine is a CPU emulator framework, based on QEMU, that is still in its infancy (on Version 1.0.1). It targets many different architectures (ARM, ARM64, MIPS, X86, M68K, and SPARC), whereas most CPU emulators focus on single architectures. However, emulating in Unicorn is very much a non-trivial task. We chose to use Python for our emulation harness because of our familiarity and the multitude of Python examples for both Unicorn and afl-unicorn.

We started by exploring some of the tutorials[15] and examples for using Unicorn and were able to successfully emulate small trivial binaries in various architectures (MIPS, ARM, x86). Once we had some working examples, we used the ARM example as a basis to build off of and attempted to emulate FreeRTOS. Unfortunately, we had trouble emulating FreeRTOS successfully. Unicorn would always throw errors about invalid memory reads and writes, but the perplexing part was that these reads and writes were in mapped addresses.

The BBB has RAM from address 0x80000000 to 0x9FFFFFFF (512MB), and in Unicorn we would map 512MB of memory starting at address 0x80000000. Every single "unmapped address" Unicorn complained about during a crash was within this mapped range. We tried mapping less using the program size in bytes, page aligned to 4Kb, but this made the emulation stop even sooner. There is obviously much we still do not understand about how Unicorn partitions memory and this halted our progress on emulating FreeRTOS. We did notice that Unicorn seems to crash when executing a **bx** or **blx** instruction which leads us to believe it has some bugs when doing a context switch between ARM and Thumb.

Since we could not get FreeRTOS emulated successfully we tried a simple bare metal program shown below:

```
#include <string.h>
#include <stdlib.h>
#include "soc_AM335x.h"
#include "beaglebone.h"
#include "consoleUtils.h"
int main()
{
   char* param = (char*)0x80030000;
   int x = 0;
   /* Initialize console for communication with the Host
        Machine */
    ConsoleUtilsInit();
    ConsoleUtilsSetType(CONSOLE_UART);
    ConsoleUtilsPrintf("Hello from Beaglebone Black\n\0");

   //copy message
   strncpy(param, "Hello World\r\n\0", 14);
   while(1)
   {
      ConsoleUtilsPrintf(param);
      for(x = 0; x < 100000000; x++); //busy wait
   }
   return 0;
}
```

This program would copy a string message to an address location and use a UART to print it continuously to a serial console. This is the same program that we used to test the OpenOCD script. This is a simpler program that FreeRTOS by orders of magnitude, but still much more complicated than the simple binaries we had success emulating. The addition of a UART would have allowed us the opportunity to test the JTAG forwarding to read/write to the UART control and status registers. However, we encountered more issues with this binary than we did with the FreeRTOS binary. During emulation, this binary is not read correctly by Unicorn. Every instruction is **andeq r0, r0, r0** which corresponds to an instruction where all bits are 0. However, when looking at the binary file with **objdump**, we can easily see that this is not the case. We had not changed anything about our emulation setup other than the filename of the binary, and we made sure the file was opened correctly without error. We are still not sure why Unicorn can not read this binary but can read other binaries built with the same tool chain.

III. Future Work

Unfortunately, we have not had any much success in bringing all of the components together. The main source of our difficulty is with the Unicorn Engine. We have been struggling to get binaries to emulate without crashing—even small bare metal code. There is much we still do not understand about how Unicorn works which is mostly due to its lack of documentation. As such, we have not been able to get any code to emulate far enough to need access to IO.

Because of this, there is still much work to be done for Almiraj to be a useful method of fuzzing embedded systems. In order of importance, the following work should be completed:

1) Create a better emulation script that can emulate full binary files without crashing.

2) Add support to the emulation script for IO forwaring, extending our JSON memmap file to include IO peripherals.
3) Fuzz our emulated binary with our IO forwarding extensions built into the emulation
4) Improve IO performance using USB3.0 bridge similar to Inception[4].
5) Incorporate a stronger fuzzer, e.g. CollAFL[10].

While we did not succeed in creating a complete tool, we feel that the work we have done has laid a good foundation for developing a tool to fuzz arbitrary embedded devices.

REFERENCES

[1] N. Voss, "afl-unicorn: Fuzzing arbitrary binary code," Oct 2017. [Online]. Available: https://hackernoon.com/afl-unicorn-fuzzing-arbitrary-binary-code-563ca28936bf

[2] ——, "afl-unicorn: Part 2 - fuzzing the 'unfuzzable'," Nov 2017. [Online]. Available: https://hackernoon.com/afl-unicorn-part-2-fuzzing-the-unfuzzable-bea8de3540a5

[3] 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, Network and Distributed Systems Security Symposium, 18-21 February 2018, San Diego, CA, USA*, San Diego, UNITED STATES, 02 2018. [Online]. Available: http://www.eurecom.fr/publication/5417

[4] N. Corteggiani, G. Camurati, and A. Francillon, "Inception: System-wide security testing of real-world embedded systems software," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 309–326. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/corteggiani

[5] K. Koscher, T. Kohno, and D. Molnar, "SURROGATES: Enabling near-real-time dynamic analyses of embedded systems," in *9th USENIX Workshop on Offensive Technologies (WOOT 15)*. Washington, D.C.: USENIX Association, 2015. [Online]. Available: https://www.usenix.org/conference/woot15/workshop-program/presentation/koscher

[6] B. Ghena, W. Beyer, A. Hillaker, J. Pevarnek, and J. A. Halderman, "Green lights forever: Analyzing the security of traffic infrastructure," in *8th USENIX Workshop on Offensive Technologies (WOOT 14)*. San Diego, CA: USENIX Association, 2014. [Online]. Available: https://www.usenix.org/conference/woot14/workshop-program/presentation/ghena

[7] Z. Zhang, Z. Lv, J. Mo, and S. Niu, "Vulnerabilities analysis and solution of vxworks," in *2nd International Conference on Teaching and Computational Science*. Atlantis Press, 2014. [Online]. Available: https://doi.org/10.2991/ictcs-14.2014.24

[8] T. Technologies, "De main boards - cyclone - de0-nano development and education board." [Online]. Available: https://www.terasic.com.tw/cgi-bin/page/archive.pl?No=593

[9] ——, "De main boards - cyclone - de0-nano development and education board." [Online]. Available: https://www.terasic.com.tw/cgi-bin/page/archive.pl?No=593

[10] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *2018 IEEE Symposium on Security and Privacy (SP)*, May 2018, pp. 679–696.

[11] P. E. Black, "Juliet 1.3 test suite: Changes from 1.2," *NIST Technical Note 1995*, Jun 2018.

[12] Unicorn, "Unicorn." [Online]. Available: https://www.unicorn-engine.org/

[13] J. Sandin, "Exploiting memory corruption vulnerabilities on the freertos os," Dec 2017. [Online]. Available: https://shmoo.gitbook.io/2016-shmoocon-proceedings/bring_it_on/01_exploiting_memory_corruption

[14] "Altera nios ii freertos demo." [Online]. Available: https://www.freertos.org/FreeRTOS-Nios2.html

[15] Z. Várnagy, "Avatao tool tutorials: Unicorn." [Online]. Available: https://platform.avatao.com/paths/8e720072-9169-4d4c-9569-c330ce7fd947/challenges/28f5ae81-6a01-11e6-bdf4-0800200c9a66