Analyzing Peripheral Interupts for Use in Embedded System Emulation

Fernando Maymi and Casey McGinley

Abstract— **At present, security is decidedly lacking in embedded devices. Dynamic analysis can a powerful tool for exposing vulnerabilities and improving the security of these devices in general. Unfortunately, such analyses are often bottlenecked by the cost and scalability factors of dealing with physical hardware. We believe the solution is full system emulation, facilitated by high level modeling. We take a first step towards addressing this challenge with a tool that explores a simple firmware image via symbolic execution and maps interrupt numbers to the interrupt handlers that they trigger. We believe that this tool can be generalized further to work on a broader range of firmware images of varying degrees of complexity.**

I. Introduction

Embedded devices are woefully insecure and vulnerable, and there are several studies in the current literature which corroborate this statement. For example, in 2014 a French research group used basic symbolic execution techniques to discover 38 new vulnerabilities in 693 firmware samples [8]. Additionally, in 2013 a research group at Columbia demonstrated a general purpose methodology for leveraging vulnerable firmware update mechanisms on many embedded devices [9]. As such, there is need for new and effective methods to analyze firmware images dynamically. We believe our tool represents one of the first steps towards an effective and generalized solution for emulating embedded devices, ultimately facilitating the dynamic analysis of embedded firmware images.

A. Current Techniques for Analyzing Embedded Systems

The classical way of dynamically analyzing an embedded device requires the firmware to be executed on the hardware itself, with a hardware debugger attached. Analyzing firmware images in this way is limited in several capacities. First of all, hardware debuggers are difficult to work with, and especially difficult to integrate into existing dynamic analysis platforms. Furthermore, when working with hardware debuggers, there is the possibility that your test and analyses might bring the device into an improper state, causing damage to the hardware itself. Finally, analyzing directly on hardware is an unscalable solution, as for every firmware image you want to analyze, you need a corresponding piece of hardware, which can get expensive and logistically problematic. Similarly, the hybrid solution Avatar [19] facilitates the use of existing dynamic analyses on firmware images, but since the hardware is still used partially, it is not a scalable solution and may cause damage to the hardware.

B. Embedasploit

Embedasploit is an attempt to develop a "Red-Team in a Box" for industrial control systems (ICS). ICS are built on the embedded systems we have discussed. As such they suffer from the same deficiencies in their security. These vulnerabilities can be dangerous as ICS are used in a variety of critical infrastructure systems, including power and water treatment plants. Embedasploit aims to tackle these vulnerabilities by making penetration testing of industrial control systems easier. Embedasploit largely consists of three components: a fingerprinting component, a vulnerability library and an exploit payload system.

Our work fits into that second component. Building that vulnerability library, while possible to do using the current methods discuss in Section I-A, would be facilitated by a more scalable approach such as whole system emulation. Whole system emulation doesn't suffer from problems of scale (or potential hardware damage), because the physical hardware is removed from the equation entirely. Additionally, in an emulated environment, everything is implemented as software, meaning that anything can be modified or edited, allowing for easy instrumentation and dynamic analysis of firmware images.

To further increase simplicity and speed, we can punt on complex peripheral devices by using high level modeling. High level modeling replaces a software-hardware interface with a simple simulation. This re-

duce complexity but makes that interface unavailable to analysis. Our project takes a modest step towards this type of high level modeling by discovering information about how the firmware will respond to interrupts from peripherals.

C. Concolic Testing and angr

Our tool uses angr [18], a concolic testing system, to map interrupt numbers to their handlers. Concolic testing combines concrete testing, which runs and tracks the performance of a program, with symbolic execution. Symbolic execution is a powerful technique which tracks the parameters of possible values through the execution of a program. This allows us to analyze firmware without all the necessary peripheral devices, by replacing their input with parameterized variables. Concolic execution combines these two approaches, using symbolic execution to find possible paths and generate inputs to concretely execute along. Angr provides a wide array of analytical capabilities, from static to dynamic, and from purely symbolic to concolic. It is also implemented as a python module, allowing it to be built into a script quickly and easily.

D. Econotag

The Econotag [13] is an open source and exceptionally simple example of an embedded system. It is built on MC13224v, an ARM7 micro-controller with an integrated radio. The fact that Econotag uses ARM poses a minor problem as there are a couple instruction that angr does not handle, fortunately it is easy to hook a function to those instruction, solving the problem. The Econotag has very few peripherals and it is surprisingly well-documented, making this a good device for testing our approach.

II. Design & Methodology

A. Initial setup

In order for our tool to successfully explore and symbolically execute the Econotag firmware image, a few initial steps had to be taken. The first such step was hooking the MSR and MRS instructions with a custom function to simulate their behavior. The MRS and MSR instructions are used to read and write respectively from the ARM CPSR (Current Program Status Register) and they are not directly supported by angr. The addresses of these instructions in the firmware were found using a simple combination of objdump and grep. Then, using angrs built-in hooking functionality, we simulated these instructions using a high-level function written in Python.

In order for the script to successfully map the interrupt numbers, the interrupt handler vector table must, of course, exist in memory. Unfortunately, in most firmwares, the interrupt vector table isnt statically defined in the firmwares ROM, but is rather loaded into memory at some point during the boot process. As such, in order for our tools exploration to be successful, the interrupt vector table must be pre-loaded into memory, which means we first need to determine the addresses in the vector table, as well as its base address. By running this firmware in PANDA [10] (a dynamic analysis platform), the firmware was able to get far enough into its boot process that memory could be dumped and the interrupt handler vector table could be recovered (see Table I).

address in memory	handler addresses
0x00400120:	0x0001001d 0x00003275
0x00400128:	0x0001001d 0x00003f35
0x00400130:	0x0001001d 0x0001001d
0x00400138:	0x0001001d 0x004029d1
0x00400140:	0x0001001d 0x0001001d
0x00400148:	0x0001001d

TABLE I: Interrupt handler vector table

Finally, in addition to the base address of the vector table, several other critical addresses had to be determined ahead of time, including the address to begin executing from (e.g. the address of the function which actually loads the handler addresses and jumps to them) and the address at which the interrupt number is read from. In our case, all three of these addresses were determined by a combination of consulting the documentation and examining the firmware in the IDA Pro disassembler.

The base address, as indicated in the table above was found at 0x00400120. The function which loads handler addresses from the vector table was found to be at 0x0001082c. However, we actually start executing from 0x00010810, a function which jumps to 0x0001082c. We do this since 0x00010810 actually initializes some registers to non-zero values, and having zero-valued registers was giving us incorrect results later on in our exploration. In retrospect, we could have just started at 0x0001082c and initialized the registers to symbolic values ourselves. As for the address where the interrupt number is read from, instead of finding that explicitly, we instead identified an address (0x00010834) where register R0 is loaded with a symbolic value for the interrupt number.

B. Handler Mapping

Having found the prerequisite information described in the previous section, our tool can explore and perform its mapping. We start by initializing an angr Project and State instance. The project manages all components and acts as a factory for generating all instances of other objects (state, path, pathgroup, etc.) relevant to the firmware. The state emulates the memory for the binary, keeps track of the constraints collected as execution progresses, and gives us access to a constraint solver for the symbolic variables we encounter. We add our hooks for each MRS and MSR instruction and set the entry point to be 0x00010810, the function just before the function which actually jumps to the interrupt handlers.

Angr manages its executions through a binary using Path objects, and it allows for the management of several paths through a binary with a Path Group object. Using a Path and Path Group instance, we symbolically execute until we reach 0x00010834, the address just after register R0 has been loaded with the interrupt number. Since this is symbolic execution, the value stored in R0 is symbolic as well, representing all possible values for the interrupt number.

At this point, we store this symbolic value for later, and we also add a constraint to the state manually, namely that R0 is less than 0xB (11 in decimal). This step was necessary as angr seemingly failed to pick up the implied constraint of a CMP instruction at 0x00010834. Here, the firmware compares R0 (the interrupt number) to 0xB and only proceeds to execute four load/branch instructions if R0 was less than 0xB. We expect angr is failing to pick the constraint up due to the intermediary conditional loading instructions between CMP and BX and that it would get the constraint if these intermediary conditional instructions were all encapsulated within a separate conditional branch.

With R0 recorded and the constraint set, we allow the execution of the path to step to the next basic block, which should be one of the interrupt handlers. The execution forks off into four possible Paths at the addresses seen in Table II

successor path	address
Path 1	0x00003275
Path 2	0x00003f35
Path 3	0x001001d
Path 4	0x04029d1

TABLE II: Execution paths

For each of these new paths, we utilize their states constraint solver to concretize the value of R0 we recorded earlier in the execution. After getting a concrete value for R0 we add the constraint that R0 is not equal to the value we just solved for. Then, we concretize R0 again. We repeat this process until R0 becomes unsatisfiable, meaning that no more possible values of R0 exists. In this way, we ensure that we have found an exhaustive list of all possible interrupt numbers for each path.

Using the addresses of these paths and the values of R0 we concretized, we get the mapping of interrupt handlers and numbers shown in Table III

handler address	number
0x0001001d	0, 2, 4, 5 6, 8, 9, 10
0x00003275	1
0x00003f35	3
0x004029d1	7

TABLE III: Interrupt handler address & number mapping

III. Results

As can be seen in the previous table, our tool identified four distinct handlers in the Econotag firmware at addresses 0x0001001d, 0x00003275, 0x00003f35 and 0x004029d1. Each handler was mapped to exactly one interrupt number, with the exception of the 0x0001001d handler, which was mapped to eight interrupt numbers. This is because 0x0001001d is just a dummy function meant as a placeholder for unused interrupt numbers. This handler just returns to the caller and does nothing else. Given that we loaded the interrupt vector table into memory manually, we know that our tool has indeed found all of the expected handlers. In addition, our manual analysis in IDA Pro confirms that the interrupt numbers should range from zero to ten.

In terms of performance, the tool runs very fast on this firmware. All the timings were measured on a virtual machine with specs in Table IV.

OS	Ubuntu 16.04 LTS
Architecture	64-bit
Memory	4GB
Processor	Intel i5-4690K
Speed	3.50GHz
Cores	4

TABLE IV: Virtual machine specs.

Running the tool one hundred times, we found the average runtime to be 0.3668 seconds. In addition, our

setup procedures, on average, accounted for only 4.28% of the runtime, with the actual execution and mapping making 95.61% of the runtime. Breaking the runtime down in terms of symbolic exploration and constraint solving, symbolic exploration accounted for 51.38% of the runtime and constraint solving accounted for 44.23%. These results are summarized in Table V.

metric	value
total runtime	0.36675 s
setup runtime %	4.27726%
execution/mapping %	95.61057%
symbolic exploration %	51.37609%
constrain solving %	44.23370%

TABLE V: Performance analysis

IV. Limitations and Future Work

While the method of analysis we describe in this paper is generalizable in theory, there are a number of practical limitations on generalizing this tool. One of the most problematic of these limitations is the number of addresses that we needed to determine manually. That said, upon completion of our tool we did discover at least one portion of this manual analysis that could be automated and generalized. Instead of manually and statically instructing the tool to begin its exploration at 0x00010810, we could instead have it begin at address 0x00000018 or 0x0000001c. These two addresses are the ARM-specified FIQ and IRQ addresses and they are present in all ARM-based firmwares. Any interrupts would first go through the FIQ or IRQ handlers before being passed to the individual firmwares interrupt handlers. As such, if we begin execution at either of these addresses, we should reach 0x0001082c (or its equivalent in another firmware) in a timely manner.

Additionally the need to retrieve the vector table before using our tool makes this harder to use. We could integrate the two processes but generating the vector table is a slow process (see Section III). One way to compensate for the delay is to store the vector table after it has been generated the first time. Future uses of the tool on that firmware would load the vector table from a file if available so that the vector table would only have to be generated once.

V. Code

All of our code as well as the Econotag firmware image we used is available on GitHub at:

`https://github.com/fmaymi/Isomano`

Our tool is encapsulated by the script map_handlers.py. At the time of writing, the most recent commit to alter this file was $ace14f6fb68e17863348a989dd45fbd9d1730ffa$.
In addition, there is a branch in this repository called timing-instrumentation available at:

`https://github.com/fmaymi/Isomano/tree/timing-instrumentation`

In this branch, map_handlers.py has been modified (very messily) to time certain portions of the code to produce the metrics seen in Section 3. At the time of writing, the most recent commit altering map_handlers.py on this branch is $75766a4a70c3a931ee32a30068f2266b6829d366$.

VI. Related Work

In the very specific context of our project, we know of no previous work attempting to solve the same problem. That said, there is a large body of existing research into the dynamic analysis of embedded devices and their overall security.

The most common tool for dynamic analysis of embedded systems are hardware debuggers, particularly JTAG [1], the standard developed by IEEE. A recently proposed alternative is Avatar [19], a dynamic analysis platform designed for firmware images. Avatar makes use of an existing general-purpose dynamic analysis platform known as S^2E [6] to emulate and analyze firmware images. In order to deal with the problem of poorly specified peripherals in embedded devices, Avatar forwards all attempted interactions with the peripherals in the S^2E environment to actual hardware, leaving the hardware restrictions.

The whole system emulation of embedded systems we promote will allow us to leverage powerful existing dynamic analysis platforms like PANDA [10], which allows users to record and replay execution traces so that dynamic analysis can be performed out of sync with the execution itself, an idea first proposed by Chow et. al. at VMware [7]. PANDA's dynamic analyses are built on top of QEMU [3], an open-source emulation platform supporting many different CPU architectures.

Symbolic execution [11] is a method of analyzing a program by executing on symbolic values, values which represent valid ranges or conditions. In contrast, concrete execution is 'business as usual', executing a program on real or concrete inputs. Symbolic execution allows a user to determine parameters for inputs that result in code execution along a given path. In addition,

although often used interchangeably with symbolic execution, concolic execution [16] is actually a hybrid technique wherein concrete inputs identify a single execution trace which is then explored symbolically.

SimuVEX is angr's symbolic execution engine, which tracks the program state using a python based wrapper [17] of the intermediate representation VEX [12]. The dynamic symbolic execution is based on the techniques described in Mayhem [5] which is more memory efficient than other options, allowing us to execute the firmware long enough for it to populate things like interrupt handlers which may not be present in memory until after boot. Angr supplements Mayhem with the veritesting static symbolic execution system [2] which reduces the execution space when multiple paths would have the same result. Another feature of angr that may be useful is called Under Constrained Symbolic Execution (UCSE) [14], which allows individual functions to be analyzed, though at the cost of increased false positives. This feature is implemented in angr as UC-angr, and it is based on a previous tool called UC-KLEE [15], which itself was a UCSE enhancement on top of the dynamic symbolic execution tool KLEE [4].

VII. Acknowledgements

We'd like to thank Prof. Brendan Dolan-Gavitt. In addition to orchestrating this project and the larger project it is a part of, he gave us a considerable head start and guidance throughout. Specifically, Prof. Dolan-Gavitt provided code that generates the vector table for the econotag firmware, several of the addresses that needed to be identified manually, a unified binary containing the Econotag firmware and ROM, and hooks for MSR and MRS instructions in ARM. Some of these road blocks could have added many man-hours to our project so we are grateful for all of his assistance.

References

[1] Ieee standard for test access port and boundary-scan architecture. pages 1–444, 2013.

[2] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1083–1094. ACM, 2014.

[3] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[4] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[5] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 380–394, Washington, DC, USA, 2012. IEEE Computer Society.

[6] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: a platform for in-vivo multi-path analysis of software systems. *ACM SIGPLAN Notices*, 46(3):265–278, 2011.

[7] Jim Chow, Tal Garfinkel, and Peter M Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 1–14, 2008.

[8] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 95–110, 2014.

[9] Ang Cui, Michael Costello, and Salvatore J Stolfo. When firmware modifications attack: A case study of embedded exploitation. In *NDSS*, 2013.

[10] Brendan F Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering for the greater good with panda. 2014.

[11] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[12] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[13] pacovi. Econotags.

[14] David A. Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 49–64, Berkeley, CA, USA, 2015. USENIX Association.

[15] David A Ramos and Dawson R. Engler. Practical, low-effort equivalence verification of real code. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 669–685, Berlin, Heidelberg, 2011. Springer-Verlag.

[16] Koushik Sen. Concolic testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 571–572. ACM, 2007.

[17] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. 2015.

[18] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[19] Jonas Zaddach. Development of novel dynamic binary analysis techniques for the security analysis of embedded devices. TELECOM, ParisTech, 2015.