Towards Automated Exploit Generation for Embedded Systems

Matthew Ruffell[1], Jin B. Hong[1], Hyoungshick Kim[2], and Dong Seong Kim[1]

[1] Computer Science and Software Engineering,
University of Canterbury, New Zealand
[2] Department of Software,
Sungkyunkwan University, Seoul, South Korea
{msr50,jho102}@uclive.ac.nz,dongseong.kim@canterbury.ac.nz
hyoung@skku.edu

Abstract. Manual vulnerability discovery and exploit development on an executable are very challenging tasks for developers. Therefore, the automation of those tasks is becoming interesting in the field of software security. In this paper, we implement an approach of automated exploit generation for firmware of embedded systems by extending an existing dynamic analysis framework called *Avatar*. Embedded systems occupy a significant portion of the market but lack typical security features found on general purpose computers, making them prone to critical vulnerabilities. We discuss several techniques to automatically discover vulnerabilities and generate exploits for embedded systems, and evaluate our proposed approach by generating exploits for two vulnerable firmware written for a popular ARM Cortex-M3 microcontroller.

Keywords: Embedded System, Exploit Generation, Software Vulnerability

1 Introduction

Embedded systems are small low powered computers that carry out a specific task. To keep costs down, embedded systems typically omit modern security features such as Address Space Layout Randomisation (ASLR) or Data Execution Protection (DEP / W$\oplus$E) [17] which make exploitation of vulnerabilities significantly easier. Most software on embedded systems is also never updated or patched [12], so systems remain vulnerable even when vulnerabilities are found and disclosed. It then becomes important to find vulnerabilities in the development stage.

Unlike personal computers, conventional static and dynamic analysis tools are often ineffective in analysing firmware because non-standard peripherals are typically used in embedded systems. It takes considerable effort to emulate the behaviours of a peripheral which greatly slows the analysis of firmware. Hence, we need an efficient dynamic analysis tool which can automatically detect vulnerabilities and generate possible exploits even with specialized peripherals that the firmware interacts with.

The aim of this paper is to extend an existing dynamic analysis framework, namely *Avatar* [19], to automatically generate exploits on embedded systems. The tool was originally developed to analyse a wide range of firmware without source code since

most users have no direct access to the source code of firmware in many embedded systems. Ideally there should be limited human interaction in the vulnerability discovery and exploit generation process, to make the tool useful to even less skilled developers. All codes developed in this paper can be found in [1] [2].

The contributions of this paper are as follows:

- We extended a security analysis framework with generic device input communication and automated exploit generation modules to analyse firmware for embedded systems;
- We evaluate the feasibility of the proposed framework with two vulnerable custom firmwares.

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 presents the design and implementation of our framework. Section 4 presents the evaluation of our proposed framework. Then, discussion is given in Section 5, and we conclude our paper in Section 6.

2 Related Work

Automated Vulnerability Analysis:

The process of automatically discovering security vulnerabilities in a program is referred to generally as *automated vulnerability analysis*. Costin *et al.* [11] implemented a wide scale automated vulnerability analysis service for firmware images. Mulliner *et al.* [15] implemented a fuzzer which automatically sends randomly crafted SMS messages to mobile phones. However, their vulnerability detection engine is not intelligent, and is limited to detecting simple faults which happen to crash mobile phones. An intelligent fuzzing tool, TaintScope, has been built by Wang *et al.* [18], which bolsters fuzzing with dynamic taint analysis and symbolic execution to target fuzzing towards attacker controlled input. Davidson *et al.* [13] implemented FIE, a tool that uses symbolic execution to verify memory safety for the MSP430 microcontroller. Symbolic execution is becoming popular a mechanism to verify memory safety. Intel [5] has also started analysing the firmware for its processors using S2E [10].

Source Code Based Automated Exploit Generation: Source code based automated exploit generation tools can generate exploits with full knowledge of source code. Exploits generated are typically not very reliable as exploits may behave differently when applied to program binaries which are compiled and optimised by different compilers. Avgerinos *et al.* [3] implemented AEG, the first end-to-end system for automated exploit generation.

Binary Code Based Automated Exploit Generation: Binary code based automated exploit generation tools can generate exploits from analysing binary program distributions. Exploits are typically reliable since they are generated specifically for the program binary, but may not necessarily evade memory protection techniques of host operating systems. Automated exploit generation tools have been mainly developed for general purpose computers, as none currently exist for embedded systems. Brumley *et al.* [7] introduced a method to automatically generate an exploit by analysing a vulnerable binary program P, and the patched binary program P'. Schwartz *et al.* [17]

built Q, a tool which can automatically build ROP [16] exploits for a given binary program. Dynamic taint analysis is performed in conjunction with symbolic execution to find vulnerable program states. If the vulnerability can be exploited by ROP, then *gadgets* [16] are located in the binary and a payload generated. A similar framework, Crax, by Huang *et al.* [14] uses program crash traces as input. Crash traces can be found from typical static or dynamic analysis tools such as fuzzers, or from normal use. Crash traces are then used as execution traces for concolic symbolic execution within the S2E [10] framework, and if the crash condition is exploitable, an exploit could be produced. Cha *et al.* [9] developed Mayhem, a tool which automatically generates exploits for a given binary program, with no additional information required. Mayhem was run over all binaries in the Debian Linux distribution, and over 13,000 bugs were found and 150 exploits generated [4].

3 Proposed Approach: Design and Implementation

We extend the Avatar framework to automatically generate exploits for embedded systems firmware. First, we explain features implemented by the Avatar framework in detail in Section 3.1, then we describe the implementation in Section 3.2.

3.1 Avatar Framework

Avatar [19] is an event driven dynamic analysis framework. On a high level, Avatar is responsible for executing firmware and testing its behaviours based on the emulation of a target device. The overview of the Avatar architecture is shown in Figure 1.

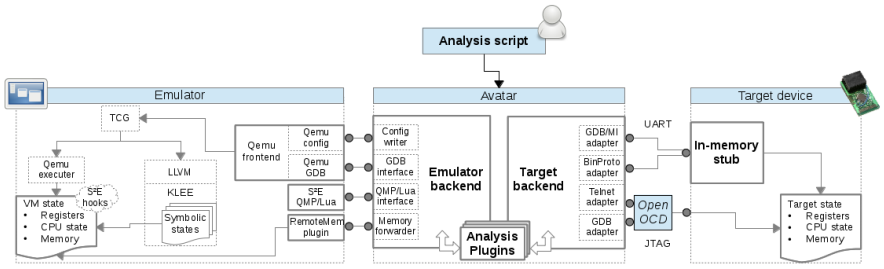

Fig. 1. An overview of the Avatar architecture (adapted from [19])

Avatar provides a concrete wrapper implementation to use the Selective Symbolic Execution (S2E) [10] framework. S2E is a flexible framework that supports emulating applications and firmware in QEMU [6] that is a machine emulator supporting many architectures (e.g., ARM, X86, MIPS and SPARC) while performing symbolic execution with KLEE [8] concurrently. I/O operations can be intercepted and forwarded to the physical device while signals and interrupts can be injected into the emulator.

3.2 Implementation

Avatar Configuration The Avatar configuration file is the core Python script that controls the operations of the Avatar framework. This file imports all relevant libraries for analysis, and contains configuration parameters required for S2E.

S2E requires considerable configuration for emulating a target device. Firstly, the hardware of the target device needs to be specified in order to create a virtual machine that closely emulates the target processor. Memory ranges also need to be mapped manually, according to the layout of the target device. This is to ensure that the addresses contained in the firmware match with those on the emulator, and memory regions which can be marked as local to the emulator are so. At a minimum the code and RAM regions should be mapped to the processor. Avatar will then forward any operations that involve addresses outside of those regions to the target device. If the code and RAM are not mapped, then all memory operations will be forwarded to the target device.

Plugins that are loaded directly into S2E must also be configured. The most notable plugins are the RawMonitor, ModuleExecutionDetector and Annotation. RawMonitor simply assigns memory regions to modules. ModuleExecutionDetector then keeps track of the program counter in relation to modules, and calls any plugins which register dependency on particular modules. Annotation allows the user to call Lua callback functions to exhibit symbolic execution when a particular address inside of a module is reached.

Custom functions that are too specific to be placed into the framework are also implemented inside the Avatar configuration file. These include call monitors, memory and register state transfer functions. Transferring registers is a specific implementation issue since different ARM processors have different amounts of registers outside of the mandated 12 general purpose registers. Many have different names on different processor families, and provide slightly different behaviour. For example, standard ARM processors have a Current Program Status Register (`CPSR`). This is where conditional flags are stored such as zero, negative and overflow. However, the Cortex-M3 ARM processor implements this in the `xPSR` register, and omits the `CPSR` register. Meaning that registers need to be manually defined in the actual register transfer functions in the configuration file. This also allows for convenient modification of tricky registers and flags, such as the Thumb bit in the `CPSR` / `xPSR`.

The remainder of the Avatar configuration file implements the analysis logic. This involves setting up the OpenOCD connections and loading them into the Avatar framework. Each state of the flowchart shown in Figure 2 represents one or a small group of function calls in the Avatar configuration file.

Generic Device Input Communication Most frameworks (including Avatar) have no way to communicate with the target device over its real communication channels. If input is needed to be injected into the target device, a debugger is typically used to modify the contents of received data to the injected data. However, the problem is that if exploits are injected into the firmware with a debugger, there is no way of verifying that the injected exploit is really what is sent over real communication channels. That is, we need to assure the integrity of injected data. Take a UART serial port for an example. The data to be injected to the firmware could contain machine codes that are

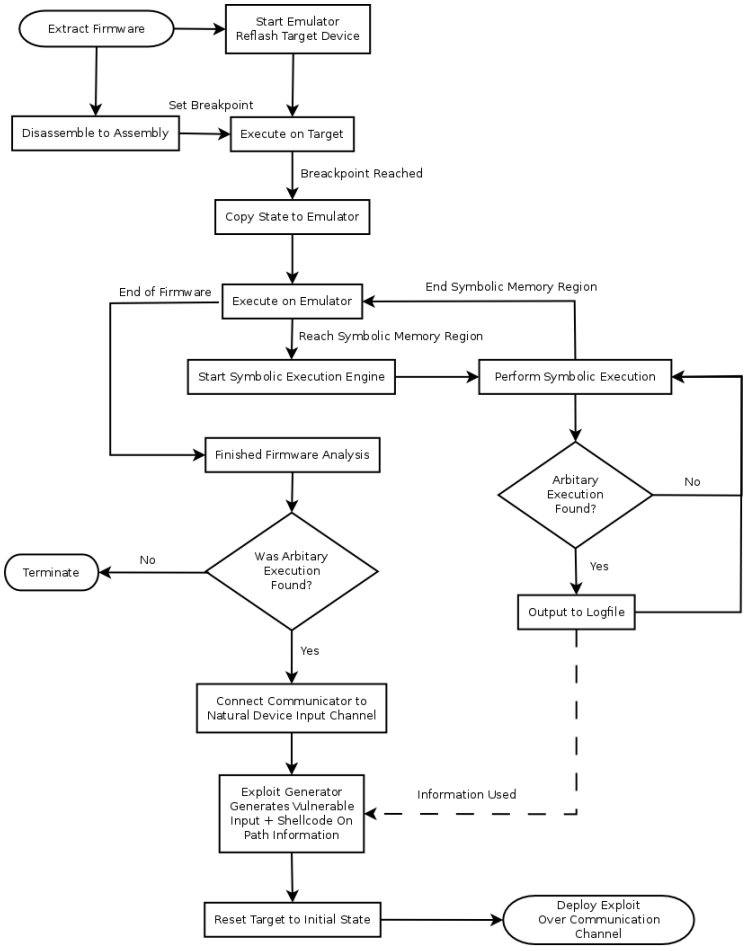

Fig. 2. Methodology behind automated exploit execution on embedded systems

interpreted as ASCII codes for newline and/or carriage return characters. When those data are injected into the firmware via a debugger, all bytes would be loaded into the firmware exactly as contained in the data. However, if those data was to be sent over a real UART serial channel, the UART transmitter would typically interpret the bytes that map to ASCII carriage return characters to indicate the end of transmission. This would cause only parts of the data to be copied, not assuring the integrity of injected data.

To overcome this problem, we developed the Communicator module, which presents a generic interface of abstract functions for implementing channel initialisation, connection, disconnection, reading and writing (which can be found in [1]). The user can simply extend the Communicator class to provide concrete implementations of abstract functions for a specific channel type, making the Communicator class suitable for any communication channel mechanism, such as Ethernet, USB, Bluetooth or serial UART.

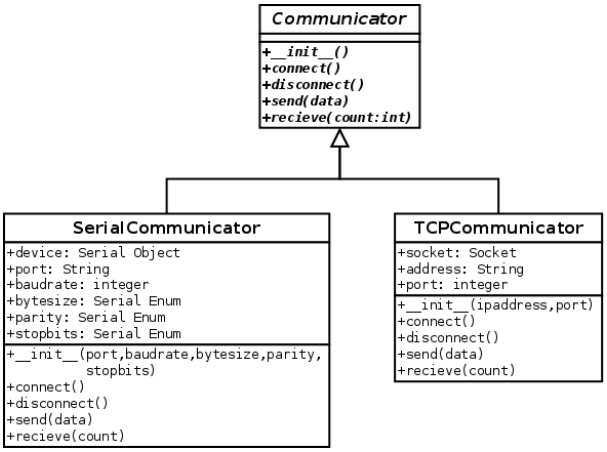

Fig. 3. UML depicting the Communicator module

Since embedded systems receive input from various sources, many concrete communicators may be active at any time. All communicators adhere to the same interface, which enables the developer to quickly and easily switch between different input channels for deploying exploits. The UML diagram in Figure 3 shows the functionalities of the Communicator module.

Exploit Generation We develop the ExploitGenerator module to automate exploit generation. To extend this module to various exploit kits, the ExploitGenerator class presents a generic interface which can be integrated with any exploitation method such as stack buffer overflow, return oriented programming, use after free and null pointer dereference. Figure 4 shows the functionalities of the ExploitGenerator module in the UML diagram form.

The ExploitGenerator module revolves around the notion that an exploit is the concatenation of an input string which places the device into a vulnerable state, and shellcode which acts upon the vulnerable state. In order to automatically generate inputs which place the device into a vulnerable state, ExploitGenerator examines path information output from the ArbitaryExecution S2E plugin. When writing the construct_input() function, a developer must take care to arrange the variables from the path in the correct order that they appear in inputs, as depending on the exploit method selected, the order that S2E provides variables from path information may not be correct. Constructing payloads is a similar matter, as existing shellcode is combined with a referenced address to the buffer found from vulnerable path information. The user also has the option to manually override the automatically generated input and payload variables.

To deploy the exploit to the target device, the ExploitGenerator class sends the exploit down a previously created generic input communication channel, denoted by a concrete implementation of the Communicator class. Since all concrete implementations of Communicator adhere to the same interface, any ExploitGenerator can send constructed exploits down any communication channel.

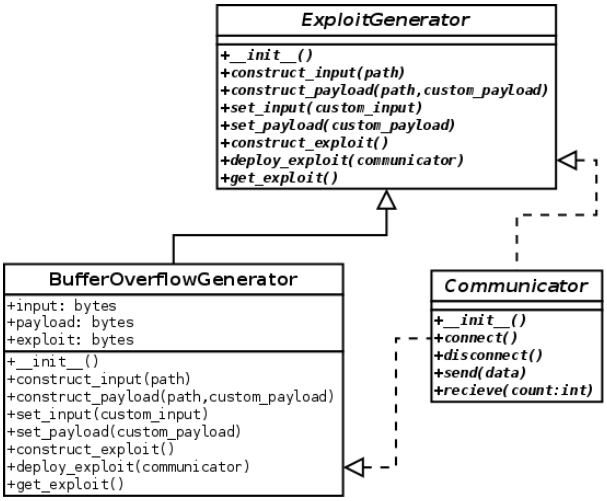

Fig. 4. UML depicting the ExploitGenerator module

For simplicity, in this paper, we only focus on automatically generating exploits for stack buffer overflow vulnerabilities. BufferOverflowGenerator is a concrete implementation of ExploitGenerator which implements this feature (which can be found in [1]). However, the developed framework is not limited to this type of attacks as it can be flexibly extended with other exploit generation modules. BufferOverflowGenerator first builds vulnerable input strings by using vulnerable path information to place the device into a state where it will read and store a buffer in a viable location. The payload is constructed such that existing shellcode is extended by a return address which points to the start of the vulnerable buffer. BufferOverflowGenerator concatenates the input and payload to generate an exploit and deploys it to the target device through a specified communication channel.

4 Evaluation

The embedded system used for evaluation is the Texas Instruments Stellaris EKS-LM3S1968 Evaluation Kit, developed by Luminary Micro. The evaluation kit features the LM3S1968 ARM Cortex-M3 embedded microprocessor, which boasts a maximum frequency of 50 MHz, 256K of onboard flash memory, and 64K of SRAM. Device debugging can be performed over USB with the popular FTDI 2232D chip, which implements USB to serial UART channels, which can be used to directly access and program the onboard flash memory. JTAG access is also provided.

The communication channel between the Stellaris board and the Avatar framework was achieved over a serial UART line, which is popularly used in real world embedded systems. In order for the host computer to communicate with the target device, an external USB UART TTY was required. A generic off-the-shelf adapter was selected to support the CP2102 UART chip.

The host computer running the Avatar and S2E frameworks has the following specifications: a 3.4GHz Quad-core Intel i7-4770 processor, 16GB of DDR3 RAM, running the 64-bit Debian 7.8 Linux distribution.

4.1 Vulnerable firmware

Two vulnerable firmware versions were developed to show the feasibility of our implementation. Those firmware versions share the common vulnerability but are significantly different in code size. Each of the firmware versions tested utilise two different hardware peripherals, a serial UART and the OLED display. Each of these peripherals must be initialised during initial device setup, even if they are not explicitly used in later stages of firmware execution. This enables the driver objects to be linked with the firmware during compilation, enabling access to those peripherals by any shellcode executed. The firmware developed share a common vulnerability that can be exploitable on some execution paths of the firmware. The vulncpy() function introduces a simple stack buffer overflow vulnerability since it does not perform any length checking of an array passed as a parameter. The vulncpy() function is called after the firmware receives a message over the serial UART line, which contains tainted data which is entirely attacker controlled. In this section, we first briefly describe how those firmware versions work and discuss the annotations for symbolic execution.

Small *Small* simply initialises hardware peripherals, receives a message over the serial UART line, and immediately passes the message buffer to vulncpy() to potentially trigger the vulnerability. *Small* receives the message by first reading in a single byte, and converting the byte from ASCII to an integer. This becomes the `length` of the buffer to receive. It then proceeds to read and fill the message buffer with `length` bytes received over the serial UART line.

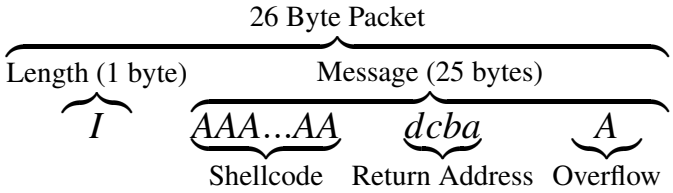

Fig. 5. Small packet structure and example exploit

The inputs required to place *Small* into a vulnerable state is simply a `length` value greater than 20, in order to overflow the buffer found in the vulncpy() function. Since there is only one path through the firmware, this is easily found with symbolic execution. The message presented in Figure 5 consists of 20 bytes of shellcode to fill the buffer in vulncpy(), 4 bytes to overwrite the return address to the desired value `0xabcd` (represented in little-endian form) and a further byte which overwrites the previous stack frame.

Large *Large* recreates an in-vivo example of a real world firmware with a complex message passing system, which can craft and display messages sent and received from the Stellaris board. The application contains 5 different views that the user can directly interact with. The application contains a significant amount of control flow logic and various nested loops and other tricky components such as dynamic memory allocation to the heap. The details of *Large* can be found in [2].

Large can be placed into a vulnerable state by sending command 1 to print an attached message to the screen. The message is parsed and vulncpy() is called before the message is printed to the screen. The message shown in Figure 6 follows the same format used in *Small* firmware.

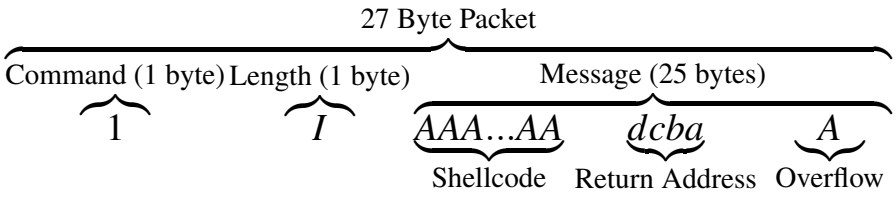

Fig. 6. Large packet structure and example exploit

Annotations Manually disassembling firmware to place annotations is one of the most time-consuming steps. Consider an excerpt of the *Small* firmware shown in Figure 7.

```
00000720 <main>:
  720:  b5f0        push    {r4, r5, r6, r7, lr} ; Context Switch
...
  744:  f000 fa85   bl      c52 <UARTCharGet>  ; Read length
  748:  b2c4        uxtb    r4, r0             ; r4 = length message
  74a:  3c30        subs    r4, #48            ; Correctly zero length
  74c:  dd0a        ble.n   764 <main+0x44>    ; if < 0 do not read
  74e:  466f        mov     r7, sp             ; r7 is buffer location
  750:  eb04 060d   add.w   r6, r4, sp         ; Allocate length bytes
  754:  4628        mov     r0, r5             ; CALL ANNOTATION HERE
  756:  f000 fa7c   bl      c52 <UARTCharGet>  ; Read 1b of message
  75a:  1e64        subs    r4, r4, #1         ; Decrement counter
  75c:  f807 0b01   strb.w  r0, [r7], #1       ; Store 1b in buffer
  760:  d1f8        bne.n   754 <main+0x34>    ; Loop and read more
```

Fig. 7. Example of annotations in the *Small* firmware

Annotations need to be placed at sections of the firmware where variables or buffers of tainted data are required to be marked symbolic. In the above example, one variable and one buffer needs to be marked symbolic. The `length` variable can be marked as

symbolic by setting the register r0 to a symbolic value since the UART driver library places a received character into register r0, a common return value register. For the buffer, the variable which points to the buffers location in memory is stored in register 7, as seen at 0x74e when the location takes the value of the stack pointer. The buffer is allocated upon the next instruction 0x750. This adds the buffer length to the current stack pointer, placing the address of the end of the buffer in register 6. The buffer consists of the bytes between the addresses of r7 and r6. The idea is to call a Lua callback function to mark those addresses as symbolic before instruction 0x754 is executed. An instruction annotation is used, which calls the required function when the program counter reaches the address 0x754.

```
function buffer_symbolic_all (state, plg)
  print ("[S2E]: making buffer symbolic\n")
  buff = state:readRegister("r7") -- r7 contains buffer address
  length = state:readRegister("r4") -- r4 contains length
  for i = 0,length do
    state:writeMemorySymb("VulnString", buff+i, 1) -- mark symbolic
  end
  -- Write null byte
  state:writeMemory(buff + length, 1, 0)
end
```

Fig. 8. Annotation callback function marking buffer as symbolic

The annotation callback function marking buffer as symbolic (see Figure 8) takes part inside of S2E, during symbolic execution with KLEE. KLEE reads the address and length of the buffer from the registers of the emulator, and then iteratively marks each byte as symbolic. By utilising similar annotations, two firmware versions were tested with our implementation.

4.2 Exploits Generated

Generating exploits for *Small* is a straightforward process with the extended Avatar framework. The vulnerability is triggered if there are more than 20 bytes copied into the buffer, which means that the first length character must be greater than 20. Since *Small* length in as a printable ASCII character, the length is offset by the character '0' , or 0x30. This means that the SMT solver was tasked to find values greater than 20 which include the offset. Two exploits are shown in Figure 9. In both exploits, the length value satisfies the minimum value of 0x44 (20). Note that the shellcode used is a string of 'a' (0x61) characters acting as placeholders, and the address of the buffer is always the same. If a debugger is consulted at run time, the buffer is allocated between 0x200000B8 and 0x200000D1, which agrees with the generated exploits.

Unfortunately, our implementation failed to automatically generate an exploit for *Large* within reasonable time. This is because a significant amount of execution paths were explored with various input mechanisms (e.g., push buttons, UART packets, etc.).

```
Length Shellcode                                        Return   Overflow
4B     61616161616161616161616161616161616161616161 b8000020 61
7F     61616161616161616161616161616161616161616161 b8000020 61
```

Fig. 9. Exploits generated for Small, shown in hexadecimal form

Moreover, symbolic execution often generated states to explore already explored paths, which finally leaded to a state space explosion in the search field.

5 Discussion

We found that Avatar is not scalable for large complex embedded systems, as demonstrated in Section 4. The action of performing symbolic execution and passing all memory-mapped I/O peripheral accesses over USB to the target device is too slow for real systems. For example, firmware for a baseband processor used in cellular phones is typically several megabytes in size, and utilise many nested loops and state machines to implement the GSM protocol. Since symbolic execution would likely tend towards state space explosion, time critical radio peripheral accesses can also fail, and the USB debug channel may exhaust bandwidth to cope with the interrupts generated.

Moreover, Avatar has no mechanism to determine what class of vulnerability has been detected. Avatar simply detects vulnerabilities if a symbolic variable is used as a control flow jump address (i.e., a symbolic variable is loaded to the program counter). A method to automatically distinguish between various vulnerability classes and a feature needs to be implemented in the extended Avatar which would automatically select the required ExploitGenerator. This would remove another decision users need to make when setting up the Avatar configuration file, as it may not be known what class of vulnerability is inside the firmware under test.

We did not evaluate our framework with real world scenarios. However, as we described previously, our custom built firmware allow us to comparatively analyse how our proposed framework can generate exploits for different complexity of firmware. But for our future work, we will investigate the effectiveness of our proposed framework for various types of real world embedded systems firmware.

6 Conclusion

Embedded systems often lack capabilities to support security features. Hence, in embedded systems, finding security flaws is essential in their development stages. In this paper, we extended the Avatar framework to implement an automated exploit generation tool for embedded systems. To show the feasibility of the implemented tool, we used two independent firmware versions that share the same vulnerability but are significantly different in size.

In our experiments, the small-sized firmware was quickly exploited while we failed to automatically generate an effective exploit on the large-sized firmware. This is because a lot of execution paths were inherently generated based on the symbolic execu-

tion technique. To overcome this limitation, we will explore various heuristics to prioritize the most important execution paths when it is not feasible to consider all possible execution paths in a target firmware.

References

1. Automatic exploit generation for embedded systems - extended avatar, `https://github.com/msr50/avatar-python`
2. Automatic exploit generation for embedded systems - vulnerable firmwares, `https://github.com/msr50/avatar-stellaris`
3. Avgerinos, T., Cha, S.K., Hao, B.L.T., Brumley, D.: Aeg: Automatic exploit generation. In: NDSS. vol. 11, pp. 59–66 (2011)
4. Avgerinos, T., Cha, S.K., Rebert, A., Schwartz, E.J., Woo, M., Brumley, D.: Automatic exploit generation. Communications of the ACM 57(2), 74–84 (2014)
5. Bazhaniuk, O., Loucaides, J., Rosenbaum, L., Tuttle, M.R., Zimmer, V.: Symbolic execution for bios security1 (2015)
6. Bellard, F.: Qemu, a fast and portable dynamic translator. In: USENIX Annual Technical Conference, FREENIX Track. pp. 41–46 (2005)
7. Brumley, D., Poosankam, P., Song, D., Zheng, J.: Automatic patch-based exploit generation is possible: Techniques and implications. In: Security and Privacy, 2008. SP 2008. IEEE Symposium on. pp. 143–157. IEEE (2008)
8. Cadar, C., Dunbar, D., Engler, D.R.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI. vol. 8, pp. 209–224 (2008)
9. Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D.: Unleashing mayhem on binary code. In: Security and Privacy (SP), 2012 IEEE Symposium on. pp. 380–394. IEEE (2012)
10. Chipounov, V., Kuznetsov, V., Candea, G.: S2E: A platform for in-vivo multi-path analysis of software systems, vol. 39. ACM (2011)
11. Costin, A., Zaddach, J., Francillon, A., Balzarotti, D., Antipolis, S.: A large-scale analysis of the security of embedded firmwares. In: USENIX Security Symposium (2014)
12. Cui, A., Costello, M., Stolfo, S.J.: When firmware modifications attack: A case study of embedded exploitation. In: NDSS (2013)
13. Davidson, D., Moench, B., Ristenpart, T., Jha, S.: Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In: USENIX Security. pp. 463–478 (2013)
14. Huang, S.K., Huang, M.H., Huang, P.Y., Lai, C.W., Lu, H.L., Leong, W.M.: Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations. In: Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on. pp. 78–87. IEEE (2012)
15. Mulliner, C., Golde, N., Seifert, J.P.: Sms of death: From analyzing to attacking mobile phones on a large scale. In: USENIX Security Symposium (2011)
16. Prandini, M., Ramilli, M.: Return-oriented programming. Security & Privacy, IEEE 10(6), 84–87 (2012)
17. Schwartz, E.J., Avgerinos, T., Brumley, D.: Q: Exploit hardening made easy. In: USENIX Security Symposium (2011)
18. Wang, T., Wei, T., Gu, G., Zou, W.: Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In: Security and Privacy (SP), 2010 IEEE Symposium on. pp. 497–512. IEEE (2010)
19. Zaddach, J., Bruno, L., Francillon, A., Balzarotti, D.: Avatar: A framework to support dynamic security analysis of embedded systems firmwares. In: Proceedings of the 21st Symposium on Network and Distributed System Security (2014)