Applying Bytecode Level Automatic Exploit Generation to Embedded Systems

October 16, 2015

Matthew Ruffell
msr50@uclive.ac.nz
Department of Computer Science and Software Engineering
University of Canterbury, Christchurch, New Zealand

Supervisor: Dr DongSeong Kim
dongseong.kim@canterbury.ac.nz

Abstract

Finding vulnerabilities in software is a difficult task, typically undertaken by experts. Developers have little of the required knowledge to find complex vulnerabilities in their software products before release. Automation of vulnerability discovery and proof of concept exploit generation is key to enable developers to check and fix software vulnerabilities in the development process. Research in this field is currently directed at automatically generating exploits for software developed for general purpose computers. Embedded systems occupy a significant portion of the market and lack typical security features found on general purpose computers. In this report, we implement automatic exploit generation for embedded systems firmwares, by extending an existing dynamic analysis framework called Avatar. We discuss several techniques to discover vulnerabilities and generate exploits, and evaluate our solution by generating exploits for three vulnerable firmwares written for a popular ARM Cortex-M3 microcontroller.

Acknowledgements

I would like to sincerely thank Jonas Zaddach for all his help with debugging complex issues that kept arising in Avatar. You may be on the other side of the world in France, but I greatly appreciated the email exchanges that offered clues into solving undebuggable situations.
I would like to thank DongSeong Kim, for his support through the year as my supervisor, especially as this was an ambitious project which required many whiteboard sessions discussing complex ideas.
Finally, I would like to thank my family and friends who have supported me throughout my time at university. Through your help, you have built me up to the person who I am today.

Contents

1 Introduction

1.1 Motivation

Finding vulnerabilities in a software system is a complex and difficult task. There is no question about the importance of finding vulnerabilities, as a vulnerability enables malicious actors to hijack program flow and execute their own code. Currently, many developers of software cannot search for vulnerabilities on their own. They must rely on highly specialised security researchers to analyse and produce proof of concept exploits. This is because finding vulnerabilities is a manual process that requires extensive skills and intricate knowledge of how software works on a low level. There are two categories of vulnerability research: statically analysing program code, and dynamically executing a program and logging behaviour. Since most users only have access to binary distributions of their programs, the researcher is faced with a situation where they must either statically read and analyse this binary disassembled into assembly language, or use black box techniques such as fuzzing [17] to dynamically execute and probe unknown binaries. This approach does not scale well as programs become larger and more complex, as reading raw assembly code and trying to gain a meaningful understanding of the code to find exploit vectors becomes infeasible. Similarly for dynamic analysis, no black box testing framework could possibly test all code flows throughout a program. This leads us to the idea of automating vulnerability discovery and automatically generating proof of concept exploits. If it becomes a simple process, then developers can also use these tools to find vulnerabilities, even before the program ships.

Embedded systems are small low powered computers that carry out a specific task. They are becoming increasingly common in the world, appearing in everyday devices such as payment systems and even watches. To keep costs down, embedded systems typically omit modern security features such as Address Space Layout Randomisation (ASLR) [27] or Data Execution Protection (DEP / W$\oplus$E) [27] which make exploitation of vulnerabilities significantly more difficult. Most software on embedded systems are also never updated or patched [11], so systems remain vulnerable even when vulnerabilities are found and disclosed. It then becomes important to find vulnerabilities in the development stage. What makes embedded systems different from general purpose computers is the use of specialised hardware peripherals, such as credit card readers [12] or radios used in cellular phones [30] which use direct memory writes, such as memory-mapped IO for communication. These normally have large ageing codebases which, paired with the fact that systems are never updated, makes embedded systems an attractive target for exploitation without the fear attacks will be patched and no longer viable. Additionally, there is usually no underlying operating system, so if a device is exploited, the whole device is taken over, not just the running process, like in traditional personal computers. The problem embedded systems researchers face with having specialised hardware peripherals, is that common static and dynamic analysis techniques either fail or are ineffective in analysing firmwares that rely on non standard peripherals. It takes considerable effort to emulate or otherwise simulate the behaviour of a peripheral which greatly slows analysis. Hence, there is a need for a dynamic analysis tool which can automatically detect vulnerabilities and generate proof of concept exploits, which are aware of specialised peripherals that the firmware interacts with.

1.2 Project Goal and Restrictions

The aim of this project is to extend an existing dynamic analysis framework for embedded systems to implement automatic exploit generation. Ideally there should be limited human interaction in the vulnerability discovery and exploit generation process, to make the tool useful to non skilled developers. The tool should be able to analyse firmware bytecode, and should have no need for source code to be present. This ensures that the tool is useful to the widest audience, as most researchers have no direct access to source code of the firmware under analysis. All modifications to the existing framework should be scalable and easily extensible to a wide range of hardware and vulnerability classes.

To manage complexity for this project, we will only consider basic stack buffer overflow vulnerabilities. The firmwares used in evaluation will be small and simple, and the overall goal is to prove that automatic exploit generation is possible for simple firmwares running on a real world device.

1.3 Report Layout

The Background chapter covers three major techniques that will be consistently referenced in the report. The Related Work chapter looks at literature and the current state of research in this field. Design and Implementation looks at the Avatar framework in detail, and how the goals of this project were implemented. The Evaluation chapter assesses the implementation by exploiting three vulnerable firmwares written specifically for this project. A Discussion is presented with limitations and future work, and finally we summarise and close in the Conclusion.

2 Background

This chapter introduces concepts and techniques which are central to the problems we wish to solve and the mechanisms we used to solve them. This chapter provides a foundation of knowledge which is required to grasp future chapters. Namely, we introduce the Stack Buffer Overflow exploit class, Dynamic Taint Analysis and Symbolic Execution.

2.1 Stack Buffer Overflow Exploits

Stack buffer overflows are a class of exploit [28] which occur as a direct result of not checking array boundaries when copying data. They are the most common vulnerability found in real world software and are also the most straightforward to exploit. Stack buffer overflows operate slightly differently on ARM architecture, so this section details how they work on an ARM Coretx-M3 microprocessor using the Thumb instruction set.

Memory is typically divided up into four different regions on microprocessors, being text, data, stack and peripherals. The text region stores program code or more specifically, the firmware under consideration. The data region contains initialised data such as static variables, and space for uninitialised data, also known as the heap. The peripheral region is used for memory-mapped peripheral devices.

The stack is a continuous block of memory that contains data. A register called the Stack Pointer (SP) points to the top of the stack, and the bottom of the stack is located at a fixed address. ARM architecture allows the stack to either grow upwards, towards higher memory addresses, or downwards, towards lower address. For the purposes of this report, the stack will grow downwards.
The stack is logically organised into stack frames which are pushed to the stack upon a call to a function, and popped from the stack when the function returns [22]. Stack frames contain variables and the return address local to a function. The return address is the value of the Program Counter (PC) at the time the function is called. Typically, parameters are also pushed to the stack frame, but on ARM architecture parameters are passed to functions using registers. Consider the following function:

```
char* vulncpy(char* input) {
    char buffer[20];
    strcpy(buffer, input);
    return buffer;
}
```

By compiling to assembly code, low level stack operations are visible:

```
00000bb8 <vulncpy>:
  bb8:  b500        push    {lr}            ; Push old PC to stack
  bba:  4601        mov r1, r0              ; Move input from r0 to r1
  bbc:  f1ad 0d14   sub.w   sp, sp, #20     ; Allocate buffer on stack
  bc0:  4668        mov r0, sp              ; Set r0 to buffer
  bc2:  f000 f827   bl  c14 <strcpy>        ; Call strcpy, r0 dest, r1 source
  bc6:  4668        mov r0, sp              ; r0 holds return data, buffer
  bc8:  b005        add sp, #20             ; De-allocate buffer
  bca:  bd00        pop {pc}                ; Pop ret into PC
```

The first action a function must take is to push the value of the previous program counter to the stack frame. This is to ensure that control can be restored to the correct address after the function finishes. Since the PC is already set to the functions address, namely 0xbb8, the value of the Link Register (LR) is used, as it contains the value of the program counter before the function call. This is unique to ARM, as Intel traditionally pushes the PC to the stack frame before loading the function address into the PC.

The buffer variable is allocated at line 0xbbc, by subtracting 20 bytes from the current SP. Note that allocations must be word aligned. The Cortex-M3 is a 32 bit processor, which means that words must be allocated in 4 byte sections. Since 20 bytes is simply 5 words, there is no need to allocate additional bytes. After this operation the stack frame looks like the following:

```
top of stack    <vulncpy() frame > < main() frame         > bottom of
                 buffer       ret   previous stack frames   stack
<-----------    [          ] [  ]  [                     ]
```

If the parameter input is less than 20 bytes then strcpy() will successfully copy data into the newly created buffer. Afterwards the SP will be increased by 20 bytes to effectively deallocate buffer, and the function will terminate by popping the return address into the PC.

Stack buffer overflows occur when more data is copied to a buffer than was previously allocated, and the value of the return address is overwritten with attacker controlled code [22]. Consider the case where the input parameter is a character array which is 25 bytes long. Strcpy() will copy data until it reaches a null byte in the input array, which will leave the stack in the following state:

```
top of stack    <vulncpy() frame > < main() frame         > bottom of
                 buffer       ret   previous stack frames   stack
<-----------    [AAAAAAAAAA] [AA]  [A                     ]
```

20 bytes will be copied into buffer, with 4 bytes overwriting the return address. Additionally, 1 byte will be copied into the previous stack frame. When the function returns, the overwritten return address will be copied into the PC and that address executed as the next instruction. By overwriting the return address, an attacker effectively gains control of execution, also known as a control flow hijack [28]. Arbitrary execution can be gained by filling the input array with malicious machine code and overwriting the return address with the address of the start of the buffer array. When the function returns, control will be passed to the start of the buffer, and the malicious code executed [22].

This attack can be mitigated by using techniques such as Address Space Layout Randomisation (ASLR) [28] or Data Execution Protection (DEP / W⊕E) [28]. ASLR randomises the fixed stack address, which makes guessing the buffer address impossible, preventing the attacker from directing control to malicious code. DEP marks specific sections of memory as writeable or executable, but never both. This prevents attackers from being able to execute their shellcode after writing it. Unfortunately these techniques require an underlying operating system, as well as additional processor features, like a Memory Management Unit (MMU) [15]. Because of this, the majority of embedded systems use the same addresses for the fixed stack address, which means every time the firmware runs, the address layouts will be exactly the same. This has a side effect that the buffer in the example will always be allocated at the same address, which makes crafting shellcode simple. There are some proposed mitigations [15] that cater to the requirements of embedded systems, but these have not landed in real world toolchains.

This is the most basic form of stack buffer overflow vulnerability, known as Stack Smashing [22]. Many other variations exist, such as Arc Injection [23], where the payload stores a shell command in registers and program control is set to the System command in the C standard library, which executes arbitrary shell commands.

2.2 Dynamic Taint Analysis

The idea behind dynamic taint analysis is to monitor the flow of input through an application and assign quantitative taint values to variables depending on their source and operations applied throughout execution. Depending on when taint values are observed, it is possible to decide how much influence an attacker has over certain variables.

For programs to be useful, they need to read and process input. Taint is introduced from inputs that can potentially hamper attacker controlled data, such as a network connection that is exposed to the world. Variables which hold data from taint sources are considered *tainted* [26]. Not all input sources should be considered tainted however, as a trusted configuration file on disk would not introduce taint into the system. In this case, variables which hold clean data are considered *untainted* [26]. Immediately two problems can arise. If a dynamic taint analysis system marks a variable as tainted when it was not obtained from a taint source, the variable is *overtainted* [26]. Similarly, if the variable is obtained from a taint source and is not marked as tainted, then the variable is *undertainted* [26]. Ensuring that variables are marked correctly is important. In attack detection scenarios, an overtainted variable may raise a false positive that data is influenced by an attacker when it really is not, and similarly an undertainted variable could potentially harbour malicious code and be a false negative.

Depending on the implementation, taint can either be binary (tainted or untainted), or quantitative. In quantitative implementations, different levels of taint can be assigned to variables derived from different taint sources. For example, data which has passed simple sanity checks and type checking would have a lower taint value than data that was stored unprocessed. Typically, most implementations will treat taint as binary for ease of implementing automatic taint analysis systems and for better runtime performance.

Taint can be propagated through the program by interaction between variables. Different operations can influence taint levels differently. There are generally three types of operations: data movement instructions, data manipulation instructions and control flow instructions [21]. Data movement instructions include variable assignment, pushing and popping values from the stack and setting memory. Data manipulation instructions are typical arithmetic instructions such as addition, subtraction, exclusive or and multiplication. Control flow instructions generally change the flow of the program, such as `if-then`, `switch`, `while` and `goto`. A taint *policy* [21, 26] determines how taint is propagated throughout the program, and differs by implementation. Taint policies are typically applied to assembly instructions, since application binaries are typically processed by taint analysis programs. A standard taint policy is the following:

Operation Type	Examples	Taint Decision
Data movement	MOV, PUSH, POP, STR, LDR	Destination data will be tainted if any of the source data is tainted.
Data manipulation	ADD, SUB, MUL, DIV, XOR	Resulting data will be tainted if any of the operand data is tainted.
Control flow	JMP, B, BL, CMP	No taint is propagated. Note: condition registers may be tainted by operations, but data itself is not changed.

Table 2.1: A typical binary taint policy

Quantitative taint policies are complex by nature. Different data operations can attribute to dramatically different results, which must be taken into account when designing the taint policy. For example, using load and store instructions to concatenate tainted arrays can be useful for attackers to craft exploits, and will have a higher taint value to suit. Moreover, if each element of a tainted array would be multiplied by a constant, then taint values would be reduced, since integrity of the attacker controlled data is not ensured. Overall, a balanced system is difficult to achieve.

```
1 A = 5;
2 B = input();
3 A = A + B;
4 goto A;
```

Figure 2.1: A simple program to demonstrate taint propagation

Taint propagation can be demonstrated by applying the taint policy from Table 2.1 to the program snippet from Figure 2.1. The program snippet contains all three operation types, and shows an example of a previously untainted static variable becoming tainted through an interaction with a tainted value. See Table 2.2 for a complete explanation.

Line	Statement	Operation Type	Taint Decision
1	A = 5	Data movement	Variable A is untainted, as static assignment has no taint source
2	B = input()	Data movement	Variable B is tainted, as input is a known taint source
3	A = A + B	Data manipulation	Variable A becomes tainted due to operation with tainted variable
4	goto A	Control flow	No taint introduced, however control is handed to tainted address

Table 2.2: Applying dynamic taint analysis to Figure 2.1

There are two main scenarios where dynamic taint analysis is useful. One is where a program under test can be instrumented and executed to completion, and a log produced detailing which variables are tainted and could be influenced by attacker provided data. This allows developers to easily find and fix variables that may have inadvertently shared operations with tainted data, when they should not have. The second is to implement an alarm which activates when a control flow instruction jumps to a tainted address [21]. Applying this to the previous section, stack buffer overflows can be detected by dynamic taint analysis. Since an attacker controlled input array is copied into the local buffer which consequently overflows overwriting the return address, both the buffer and return address will be marked as tainted. When the function finishes and the return address popped into the program counter, the alarm will sound as a control flow instruction has jumped to a tainted and attacker influenced address. From there, the program could be halted or terminated since integrity is no longer assured.

There are downsides to using dynamic taint analysis exclusively. For one, it can only detect that a stack buffer overflow or similar exploit may have occurred when a control flow instruction jumps to a tainted address. The program may have been exploited previously at the start of the function [26], which could have enabled malicious actions to take place in the function body before the return address is popped. This happens because no alarm is raised when the return address is first overwritten, only when it is used to control program flow. Dynamic taint analysis is best used in conjunction with other techniques, such as symbolic execution.

Dynamic taint analysis does not require source code to be present or binaries to be modified, as programs are instrumented at run time. This enables a wide range of interesting applications. BuzzFuzz [17] is an application that applies dynamic taint analysis to find attacker controlled buffers, and then uses those buffers as fuzzing targets. Fuzzing is where random data is injected into an application, and by selectively fuzzing attacker controlled buffers, BuzzFuzz was able to create more detailed and complex test cases than random undirected fuzzing. Taintcheck [21] is an application which utilises dynamic taint analysis as a Valgrind extension to reliably detect and generate signatures for different exploit classes. Since Valgrind is a common tool used in development, developers can use Taintcheck to determine if programs contain any common exploits. Of course, this functionality is only available to applications developed for general purpose computers, as Valgrind cannot execute binaries for embedded systems which use specialised hardware peripherals.

2.3 Symbolic Execution

Symbolic execution is a dynamic process to explore all paths throughout a given program. Symbolic interpreters reduce a particular program path to a logical formula [26] which can be solved to find what inputs are required to satisfy the given path. Symbolic execution is not a new concept, as it was well explained in 1976 [19], however applications in the field of vulnerability detection and exploit generation have only become popular in the last five years.

A symbolic execution system is comprised upon a symbolic execution *engine* and a symbolic *interpreter*. The symbolic execution engine keeps track of symbolic interpreters and maintains a tree like structure of paths interpreters have explored. This interpreter can execute a unmodified program in source or binary form, depending upon the implementation. A symbolic interpreter can mark sections of memory and variables as *symbolic* [26]. Instead of variables having a *concrete* [26] value, that is, having a specific value, like number = 5; a symbol is assigned to the variable instead. This enables the interpreter to construct a logical formula of execution for a given program path. At any time the symbolic interpreter can be halted, and a Satisfiability Modulo Theories (SMT) [13] solver can be queried to construct appropriate concrete values for symbolic variables. This effectively outputs the required inputs to place the program into that particular state. If we decide to halt execution upon an unsafe memory access, we can automatically gather inputs that place the program into a vulnerable state. This idea is critical to automatic exploit generation.

Symbolic execution explores all program paths by forking symbolic interpreters at each control flow structure in the program [26]. By forking the symbolic interpreter process, each symbolic interpreter can maintain their own state information for the particular path that they are exploring, while still sharing common symbols with other interpreters that were on the same path until the branch point. It may help to visualise by thinking of the tree maintained by the symbolic execution engine. The root node is the original interpreter. At each control flow structure in the program, one or n interpreters are forked, and each node can be considered the fork point. The leaves in the tree would become the current interpreters. A common forking strategy for basic high level control flow structures are summarised in Table 2.3.

Control Flow Structure	Forking Decision
`if-then`	Fork one extra interpreter. One takes the true path, the other takes the false path
`for / while`	Fork an interpreter for each loop iteration
`switch`	Fork n-1 interpreters, each interpreter takes one branch

Table 2.3: Symbolic execution interpreter forking strategies

Typically, the further through program execution a particular interpreter reaches, the tighter the conditions the logical formula of execution place upon the symbols in that path. This is because each successive control flow statement typically introduces restrictions for a particular variable per path. For example, an `if` statement typically checks for equality or a less than greater than relationship. This additional check places additional constraints that a variable must satisfy for access to a given path. Consider the following:

```
1  int main(void) {
2      int number, square;
3      scanf("%d %d", &number, &square);
4      if (number < 10) {
5          if (number < 5 && square == 4) {
6              safe();
7          } else {
8              unsafe();
9          }
10     }
11 }
```

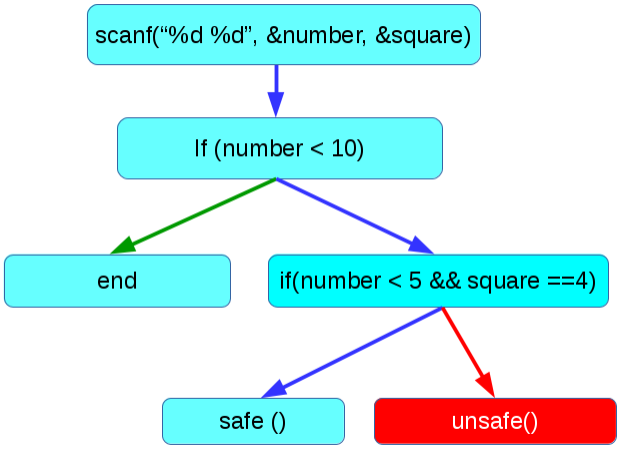

Figure 2.2: Trace of program execution with interpreters highlighted as different arrow colours

A single symbolic interpreter begins at line 2. On line 3, `number` and `square` are assigned symbols e.g., α, β since they are user provided data. Line 4 is a control flow structure, and has two possible paths. An interpreter is forked, and one takes the true path, with a constraint that $\alpha < 10$, and the other takes the false path, with $\alpha \geq 10$. The interpreter taking the false path reaches the end, and terminates. The true path interpreter reaches another control flow structure on line 5, and forks another interpreter. One takes the true path with a additional constraint that $\alpha < 5$, and $\beta = 4$. The other takes the false path, with additional constraints that $\alpha \geq 5$ and $\beta \neq 4$. During the execution of this program, a total of 3 symbolic interpreters were active, meaning that this program contains 3 possible paths. If a SMT solver is queried to generate concrete values for the third interpreter, i.e., the path that contains an unsafe memory access, it would find a value for `number` that is less than 10, but greater than or equal to 5, and assign `square` a value which is not 4. These could be used as inputs on the next run of the program to place it in a vulnerable state.

Symbolic execution can rapidly attain state space explosion [26] due to the amount of interpreters that are forked upon control flow structures. Nested `while` or `for` loops have the potential to quickly fork an exponential number of symbolic interpreters which quickly exhaust computing resources, particularly RAM. The symbolic execution engine can manage this issue by choosing different path selection strategies depending on resource usage or configuration. Three common path selection strategies are Depth First Search (DFS), Randomised and Concolic Testing [26]. DFS applies the standard DFS algorithm to the state tree managed by the symbolic execution engine. It is not usually used in practice because it can get stuck in non terminating loops, especially if a maximum depth is not specified. This causes other branches to not be explored resulting in poor code coverage. Randomised selects a leaf node from the state tree at random and continues execution of that interpreter, and halts the execution of others. Since the engine traverses the tree from the root node to select leaves, there is a bias towards shallow leaves, which means that execution may not get deep enough for satisfactory program analysis.

Concolic execution [26, 3] is where the program is first executed concretely to produce a trace of program execution. Symbolic execution then follows the concrete execution trace, and explores paths which are in the neighbourhood of concrete paths. This greatly speeds up exploration time and consumes much less resources. Further enhancements involve performing dynamic taint analysis on the program and finding buffers and variables that contain tainted data. Then during concolic execution, we can instead opt to only explore paths which use tainted data, which enables faster discovery of memory unsafe operations with attacker controlled data. From there the SMT solver can be queried to find concrete inputs that place the program in a vulnerable state, and all that is left is to provide shellcode for a functioning exploit. Similar optimisations include Veritesting [3], which on a high level switch between concrete and symbolic execution modes to significantly increase performance.

3 Related Work

This chapter presents a comprehensive survey of previous work undertaken in the field of vulnerability analysis and exploit generation. There are four main topics of interest: manual vulnerability analysis, automatic vulnerability analysis, source based exploit generation and bytecode based automatic exploit generation.

3.1 Manual Vulnerability Analysis

Manual vulnerability analysis is where a researcher manually disassembles and finds vulnerabilities in firmware binaries using traditional vulnerability discovery methods. Analysing even simple firmwares is time consuming and manual analysis is not easily scalable.

Cui *et al.* [11] disassembled firmware binaries for the HP LaserJet printer line, and performed a firmware modification attack which implemented a rootkit that allowed them to remotely command and control printers and undertake data exfiltration from the host network. Through reverse engineering Cui *et al.* discovered how to bypass quick sanity checks on software update implementations, which allowed them to install their modified firmware to the printer under attack. Since their firmware modifications was manually implemented from manually found attack vectors, it would take considerable effort to reproduce their research on a rival printing platform. Further, the techniques they used to discover vulnerabilities would be too complex for many developers to understand, meaning their research is only useful for experts in vulnerability research.

Weinmann [30] disassembled and manually reverse engineered several baseband processors from popular mobile phones. Weinmann used tools such as bindiff to find signatures of common C library functions across different firmwares, and then used traditional vulnerability discovery mechanisms to locate and construct simple stack buffer overflow exploits. Testing was performed by sending crafted baseband messages from a local transceiver station, while the target device was being locally debugged. This research has the same flaws as Cui *et al.*, as manually searching for vulnerabilities is time consuming, difficult to reproduce and not accessible to anyone other than expert vulnerability researchers.

3.2 Automatic Vulnerability Analysis

Automatic vulnerability analysis is where researchers can provide firmware as an input to a program which can then automatically discover vulnerabilities without much effort expended by the researcher. Desirable features include scalability and output pinpointing the location of the discovered vulnerability.

Costin *et al.* [10] implemented a public, wide scale automatic analysis service for firmware images. They downloaded 32,000 firmware images and unpacked them into 1.7 million binary files which they performed static analysis over. They managed to extract private RSA keys, self signed certificates, password hashes and identified potential SSH backdoors implemented in some firmwares. The most impressive part is that they can quickly update their system with knowledge of a new vulnerability, and automatically scan to find all affected devices from their pool of firmwares. Static analysis scales well as no physical device is required, and can implement some sophisticated detection engines, such as those used by Feist *et al.* [14] to detect use after free vulnerabilities.

Mulliner *et al.* [20] implemented a tool which automatically fuzzed various mobile phones by sending randomly crafted SMS messages. The tool used a local transceiver station to send randomly crafted messages to the mobile phone until the phone crashed, at which stage the message was recorded for future analysis. This experiment can easily be replicated by normal developers provided the tool, and can catch many random bugs easily and cheaply. However, vulnerability detection is not intelligent, and is limited to detecting simple faults which happen to crash the device. An intelligent fuzzing tool, TaintScope, has been built by Wang *et al.* [29], which bolsters fuzzing with dynamic taint analysis and symbolic execution to target fuzzing towards attacker controlled input. It enables more intelligent data mutation strategies to find vulnerabilities more efficiently, which resulted in finding 27 unknown vulnerabilities in popular real world software.

Davidson *et al.* [12] implemented FIE, a tool that uses symbolic execution to verify memory safety for the MSP430 microcontroller. The tool was tested by scraping Github for projects that target the MSP430, and 21 bugs were found in 99 different programs. Symbolic execution is becoming popular a mechanism to verify memory safety, as researchers from Intel [4] have also started analysing their firmware for their processors with S2E [9]. Their goal is to build a tool which allows them to analyse and generate testcases for their most complex firmwares.

3.3 Source Based Automatic Exploit Generation

Source based automatic 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.* [1] implemented AEG, the first end-to-end system for automatic exploit generation. AEG functions by first performing static analysis over the source code to gain information about the program. The program is then compiled to bytecode using GCC. Information gleaned from static analysis is used for *preconditioned* symbolic execution, which only explores paths that satisfy the precondition, and prunes paths that do not. Preconditions can be arbitrary, such as a known length of input processed, or a prefix expression, such as HTTP GET. Typically, preconditions are selected for characteristics common to existing vulnerabilities, since it increases the likelihood that such a vulnerability will be discovered by symbolic execution. If a vulnerable state is found, then concrete inputs are generated by an SMT solver to place the program into a vulnerable state. The program is then dynamically ran using the concrete inputs, and information about stack and return addresses collected. These are then used to build shellcode, which along with the concrete inputs, create a functioning exploit. AEG supports two classes of vulnerabilities: stack buffer overflows (stack smashing, arc injection) and format string (stack smashing, arc injection). AEG generated 16 control flow hijack exploits for 14 popular open source projects, proving that automatic exploit generation for general purpose computers is indeed possible.

3.4 Bytecode Based Automatic Exploit Generation

Bytecode based automatic 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. All automatic exploit generation tools surveyed below target general purpose computers, as none have been created for embedded systems.

Brumley *et al.* [6] discovered in 2008 that it was possible to automatically generate an exploit by analysing a vulnerable binary program P, and the patched binary program P'. The patched program P' would typically contain one or more sanity checks to defeat the previously unknown vulnerability in P, so their approach was to perform a diff of P and P'. This enabled them to pinpoint what code had been changed, and to use general dynamic analysis tools and dynamic taint analysis to build a logical formula of execution. This was then solved using a SMT solver to produce the required inputs to reach a vulnerable state, and depending on the type of vulnerability, output an exploit. Note – symbolic execution was not used.

Schwartz *et al.* [27] built Q, a tool which can automatically build ROP [24] exploits for a given program binary. 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* [24] are located in the binary and a payload generated. What is impressive about Q, is that it contains a robust ROP compiler which can build missing gadget types out of other available gadgets. Q can also perform exploit hardening, which given an exploit which fails due to memory protection measures, such as ASLR or DEP, Q can output an exploit which evades those mitigations. A similar framework, Crax, by Huang *et al.* [18] 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 [9] framework, and if the crash condition is exploitable, a exploit is produced. Crax also offers exploit hardening, and can mitigate the same memory protections as Q.

Cha *et al.* [8] developed Mayhem, a tool which automatically generates exploits for a given binary program, with no additional information required. Mayhem uses preconditioned symbolic execution operating in a hybrid execution mode, with a server running standard symbolic execution, and a client running concolic symbolic execution. The client first starts executing the binary concretely and performs dynamic taint analysis. If a tainted input is used for control flow, then this is reported to the server. The server then Just-In-Time (JIT) compiles the program to an intermediate language and performs symbolic execution, using information gained from dynamic taint analysis. The server keeps track of the path formula, and also maintains an exploitability formula. The exploitability formula determines if an attacker can control the instruction pointer and execute a payload. At every tainted control flow instruction, the server queries the SMT solver to solve both the path and exploitability formula. By solving the path formula, states can be pruned intelligently avoiding state space explosion, and the exploitability formula determines if an exploit can be constructed. Mayhem was run over all binaries in the Debian Linux distribution, and over 13,000 bugs were found and 150 exploits generated [2].

3.5 Avatar Framework

Zaddach *et al.* [31] developed the Avatar framework to orchestrate dynamic analysis of firmware on embedded systems. Avatar solves the problem of embedded systems having little memory and specialised peripheral devices by performing dynamic analysis inside of S2E [9] on the host computer, and forwarding all hardware requests to the target device for completion. The emulator QEMU [5], was extended to emulate ARM processors, and utilises KLEE [7], a symbolic execution engine, to perform symbolic execution over the firmware being emulated. Several devices were evaluated to test the flexibility of the framework, by performing dynamic analysis of a cellular baseband processor, a hard disk controller and a wireless sensor node.

The goal of this research project is to extend Avatar to implement automatic exploit generation of firmwares under test. Avatar is an excellent base framework to build from as it implements all techniques and technologies required to make automatic exploit generation possible, without explicitly implementing automatic exploit generation itself.

4 Design and Implementation

This chapter explains features implemented by the Avatar framework in detail, as its design and functionality significantly contribute to the overall of design of this research project. Key concepts behind the implementation of automatic exploit generation for embedded systems will be explained, as well as specific implementation details.

4.1 Avatar Framework

Avatar [31] is an event driven dynamic analysis framework, designed to ease the task of performing complex dynamic analysis on embedded systems. Avatar uses an emulator running on a host computer in conjunction with a physical target device to solve issues such as the impracticality of performing symbolic execution on the target devices tightly constrained hardware. Avatar is designed to be modular and uses a plugin-like structure, meaning that different emulators and debuggers can easily be supported by writing simple wrappers. Additional extensions are also simple to implement as a plugin, which will get called upon a specific event. Avatar comes with concrete implementations of a wrapper for the S2E [9] emulator, and the popular OpenOCD target debugger, as well all the core functionality explained below.

On a high level, Avatar is responsible for executing the firmware under test inside an emulator and on the target device, always keeping the two in sync. If the emulator requests to read a memory location which does not exist inside the emulator, such as a peripheral device for memory-mapped IO, Avatar will forward the request to the target device. If the target device receives an input or interrupt, such as an action caused by environmental factors, this request needs to be forwarded to the emulator. These actions allow the emulator to accurately represent the physical device without needing to implement or simulate specialised peripherals. Immediately, this raises questions about what device is the master, and what is the slave. The authors describe two usage scenarios, known as full-seperation mode and context switching.

Full-seperation mode [31] is typically first used when analysing new firmwares, when the researcher has little knowledge of underlying low level operations taking place. In this usage scenario, the entire firmware is executed from start to finish inside the emulator, and the emulator only has knowledge of code. The memory, or state of the emulator remains on the physical device. This in effect keeps the code and memory segments perfectly separated. Each time an instruction is executed inside the emulator, requests are made to Avatar to fetch or write data to the required address on the target device. This ensures that execution closely represents what would occur if the firmware was run natively on the target device. Further, interrupts on the target device are trapped, and forwarded back to the emulator for processing. Full-seperation mode however, is not suited to practical situations due to resource limitations. Execution inside the emulator is very slow as each memory read and write needs to be forwarded to the target device through Avatar. This brings extreme overhead, as Avatar will typically query OpenOCD to retrieve and set addresses over a USB debug connection. Execution is normally reduced to tens of instructions per second, which is prohibitively slow for medium to large firmwares. Additionally, the firmware may contain time critical sections when initialising peripheral devices, or fast polling loops such as those found in UART receive routines which must be executed at native speed.

Context switching [31] allows the researcher to begin execution of the firmware on the target device. The target device natively executes the firmware without any overheads incurred by communicating with Avatar, which is essential if the firmware contains time critical regions for peripheral initialisation, or other actions which must be performed in real time. Typically, the firmware is set to execute until a breakpoint is reached, near an interesting segment of code which the researcher wishes to more closely analyse. This also allows the researcher to quickly reach code segments of interest, without having to emulate the firmware from the absolute beginning. When a breakpoint is reached, execution is halted on the target device, and a context switch is performed. The entire state of the device (memory and registers) is then transferred to the emulator, which then resumes execution from where the device halted. Context switching can take place multiple times, as the researcher may need to view several iterations of the code segment of interest with values fetched via time critical operations. When this occurs, the emulator is halted, and any state that the emulator possess which was not kept in sync with the device (such as memory and registers) are copied to the target device. Execution then resumes from where the emulator was halted.

Avatar supports both software and hardware interrupts. Software interrupts are easy to handle, since the emulator is aware of their existence and executes their handlers immediately. Hardware interrupts however, are raised by specific peripherals to indicate that a task has been completed, or an event has occurred. These need to be trapped in the target device and then forwarded to the emulator by Avatar. Normally, interrupts are a convenient signalling mechanism that informs the firmware that say, a message has been received over UART. In this case, it is a good idea to forward the interrupt to the emulator. However, there are some interrupts that can become a nuisance during emulation, such as the periodic interrupt generated by the system clock. These periodic interrupts can easily exhaust the bandwidth over the USB debugging channel to Avatar if the frequency is high enough. They can also cause actions to happen prematurely, as the frequency they are generated at is not necessarily the speed that the emulator is executing instructions at. In this case, these interrupts need to be suppressed. Avatar provides built in provisions to drop clock interrupts on the device, and instead generate them inside of the emulator. However, it can be hard to distinguish between many periodic interrupts, as ARM only specifies that an interrupt has occurred. This causes most ARM microprocessors to include a hardware multiplexer which queries all devices to find which device generated the interrupt. This is implemented on a Nested Vector Interrupt Controller (NVIC) on the particular ARM Cortex-M3 processor used in this research project, and to suppress interrupts, the code controlling the NVIC would have to be manually disassembled to identify the correct interrupt raised. To reduce complexity in this research project, interrupts of any kind was not implemented in firmwares used in evaluation for this very reason.

There are several optimisations which exist in Avatar to speed up execution. The first is to mark sections of memory as local to the emulator. As we recall from Section 2.1, memory can be divided up into four sections being, code, stack, heap and peripherals. Avatar initially forwards all memory accesses to the device because it does not know what addresses map to which sections. To fix this, Avatar enables the researcher to mark specific ranges of memory according to their roles. Code is typically marked as read only and executable, and stack and heap is usually marked as read / writeable. Peripheral ranges are typically left unmarked so requests are still forwarded to the target device for memory-mapped IO accesses. By marking memory as local to the emulator there is significantly less communication required between the emulator and target device, which greatly speeds up execution. The other main optimisation is *selective code migration* [31] which enables functions to be called and executed on the target device, while execution is still controlled by the emulator. When a function is called, a new stack frame is pushed, and parameters are passed to the function via registers. Selective code migration allows just the stack frame and registers to be copied to the physical device, for execution of the function. It requires all exit points of the function to be replaced with breakpoints, but it enables execution to be moved from the emulator to the target device for a specific function, without copying the entire memory of the emulator back to the target device, and then back to the emulator after the function terminates, greatly speeding up execution for complex data driven firmwares.

4.1.1 S2E

Avatar provides a concrete wrapper implementation to use the Selective Symbolic Execution (S2E) [9] framework as the emulator. S2E is a very flexible framework that supports emulating applications and firmwares in QEMU, while performing symbolic execution with KLEE concurrently. S2E is extensible to support different architectures and features by means of a plugin interface. Zaddach *et al.* extended S2E to implement support for the ARM architecture, and to interface the QEMU emulator with the necessary operations required to remotely access memory on the target device. While S2E is an interesting framework in itself, this section will keep to details relevant to the Avatar framework.

S2E refines symbolic execution by selectively choosing paths to explore. In practice, this is achieved by restricting symbolic execution to specific code segments, and only allowing specific data types to be marked as symbolic [9]. This speeds symbolic execution as only areas which are interesting to the researcher are executed symbolically, as there is no longer any need to explore every single state a firmware can achieve. Further, interesting code segments typically align with areas that the researcher has selected to be executed inside the emulator, meaning that symbolic execution can be readily applied with minimal extra effort.

Typically, symbolic execution marks input sources or tainted data as symbolic, but due to how the S2E framework is constructed, the researcher must manually mark variables as symbolic. Normally this is achieved by either modifying the firmware or writing configurations that mark variables as symbolic. Modifying firmware is a poor solution as the additional instructions may make the firmware behave differently, ruining the integrity the firmware. Avatar enables researchers to write *Annotations* in Lua which specify what regions symbolic execution should take place in, and at what instructions custom Lua callback functions should be called. The custom callback functions allow the researcher to mark specific memory or registers as symbolic using a high level programming language, which allows complex logic to be easily specified in functions. By writing annotations, the firmware does not need to be modified, ensuring integrity of the firmware under test.

Zaddach *et al.* wrote a plugin for S2E to detect arbitrary execution vulnerabilities called ArbitaryExectuion. It works similarly to previously explained methods of detecting arbitrary execution with dynamic taint analysis. An alarm is raised when either a symbolic variable is used for the destination of a load or store instruction, or if the program counter or stack pointer is set to a symbolic variable. At this stage, execution of that state is halted, and a SMT solver is queried to find concrete values for variables that satisfy this vulnerable path. These are then printed to the console.

4.1.1.1 QEMU

QEMU [5] is a machine emulator which implements support for many architectures, such as ARM, X86, MIPS and SPARC. It provides full emulation of unmodified firmwares or operating systems inside of a virtual machine environment. Avatar uses QEMU inside of S2E to emulate a specific ARM processor that resides in the target device.

QEMU operates by dynamically translating instructions from the guest architecture, to an intermediate instruction set, which can be then executed on the host platform. In the Avatar usage scenario, QEMU splits regular ARM / Thumb / Thumb2 instructions down to smaller *micro operations* [5]. This process is completed by the Tiny Code Generator (TCG) at run time. Each of these micro operations are small and simple enough to represent all the different features that can be implemented across different architectures. QEMU then implements a virtual machine which can execute TCG instructions, allowing it to execute code which was compiled for a completely different architecture on the host machine. In the context of Avatar, this is what allows ARM firmwares to be emulated on a X86 based host.

4.1.1.2 KLEE

KLEE [7] is a symbolic execution engine that was initially designed to generate high coverage test cases for source code. Due to its flexibility and open source nature, KLEE has been used in many projects, and forms a core component in S2E. Avatar only communicates with KLEE to set symbolic execution regions, and register Lua callback functions. Moreover, Avatar has no other knowledge of the symbolic execution engine, making KLEE largely self contained within S2E.

KLEE monitors and maintains a tree structure which keeps track all symbolic interpreters and their forking points. Symbolic interpreters maintain their own states, in a form of a register file, stack, heap, program counter and path conditions. Path conditions are the logical formula constructed from control flow structure branch conditions which can be solved with a SMT solver. KLEE queries STP [16] as the SMT solver since STP offers a precise bit level resolution and has fast algorithms for the decision procedure.

KLEE traditionally operates by first compiling program source code to LLVM bytecode, and then performing symbolic execution over that bytecode. However, due to the applications that Avatar is used for, source code is not typically available, and the firmware is almost always compiled to machine code for the target device. To solve this issue, KLEE translates the current TCG instructions that are being executed in QEMU to LLVM instructions, when can then be symbolically executed like normal. Translation from TCG to LLVM instructions only take place when symbolic execution is active to improve system performance.

If at any stage a symbolic variable would be written to a memory address on the target device, Avatar forces KLEE to produce a concrete variable through the remote memory interface. This is to make sure that symbolic variables stay within the emulator, and that only concrete variables reach the actual hardware. Of course, there is a slight performance impact by having to halt execution for that interpreter and query the SMT solver to find an appropriate value that satisfies the path condition formula. This is a slight inconvenience, but nothing problematic enough to warrant the further issues that would arise if the physical hardware had to be able to store symbolic values.

4.1.2 Architecture and Debugging Features

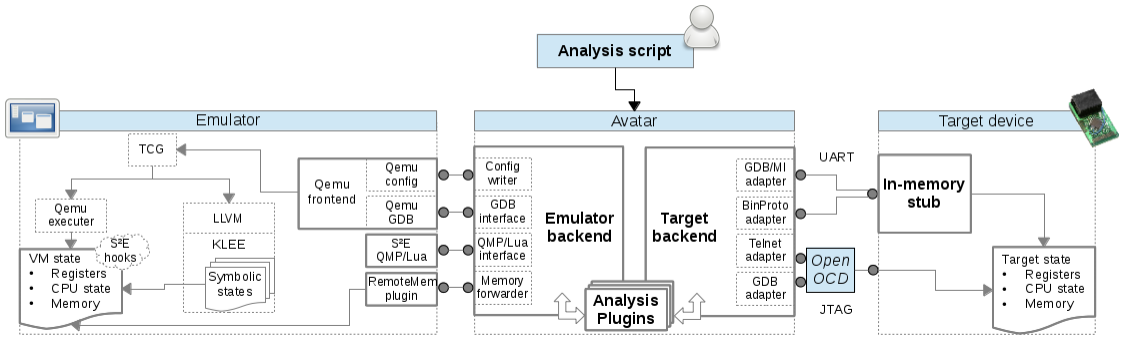

Figure 4.1: An overview of the Avatar architecture and how all components communicate [31]

Avatar was built on a slightly complex architecture that encompasses the use of many applications to be able to implement the features that Avatar provides. Figure 4.1 demonstrates this architecture and highlights the communication channels between applications. Zaddach *et al.* decided to utilise open source software throughout the implementation of Avatar, which gives the benefit that communication between applications uses simple, openly documented protocols. Further, since Avatar is a complex framework, being able to debug problems is important. Because of this, most communication protocols are human readable in plaintext, or JSON based. This section discusses how the architecture of Avatar was designed to support ease of debugging between the major components of the Avatar framework.

Avatar consists of a series of Python scripts to implement the frameworks features, and to communicate between components. Each of the scripts take advantage of a globally available logger, which documents any interesting conditions that may arise, such as the emulator failing to start, or a connection being rejected. This logger is available to custom extensions of Avatar modules, which ensures that all debugging information is available in one output stream. Since most protocols Avatar needs to communicate with are plaintext, simple sockets or telnet connections are used, which greatly reduces the complexity of debugging protocol issues.

The emulator is responsible for execution of the firmware inside of the virtual machine, symbolic execution over selected regions of code and forwarding memory access requests to Avatar. All of these actions require communication to Avatar for co-ordination.

Firstly, Avatar can control execution of the virtual machine through a GDB server supplied by QEMU. The GDB server utilises the universal GDB serial protocol, which can set breakpoints, fetch values from registers and examine memory. This channel becomes especially useful when S2E is not behaving as expected. S2E can be started manually without the Avatar framework, and the researcher can connect to QEMU directly through the GDB server. From there the researcher can set a breakpoint near a suspected failure point and single step through execution to uncover the problem. Additionally, QEMU also logs ARM / Thumb instructions that it splits during the translation process to tiny code generator instructions. This logfile is essential to review if there is any doubt that QEMU is decoding ARM instructions incorrectly[1].

Avatar communicates to KLEE during runtime through configuration files written in the Lua scripting language. Configurations include the previously discussed symbolic execution regions and annotation callback functions. These configuration files are passed to KLEE through QEMU, using the QEMU Management Protocol (QMP). QMP is a JSON based request protocol that can be easily packet sniffed to ensure that the KLEE configurations are being received and delivered correctly. KLEE also logs information about symbolic interpreters, such as what instruction each interpreter forked on, and what conditions were added to the path formula.

Each time S2E accesses a memory address, a S2E plugin called RemoteMemory catches the request, and forwards it onto Avatar through a JSON protocol. Avatar then decides to forward the access to the target device or not. All RemoteMemory requests and responses are logged to a file, which can then be replayed at a later date. This effectively enables the researcher to quickly analyse a firmware without the physical target device being present, as previous values can be replayed instead.

Avatar communicates with target devices through an appropriate debugging channel. Typically, this would be through the popular Open On-Chip Debugger (OpenOCD) [25] which can debug embedded systems through USB debugging interfaces, or JTAG. OpenOCD exposes a client interface through Telnet, which Avatar connects and sends commands to.

The Avatar configuration file allows the researcher to place print statements in between calls to the Avatar framework, or other custom memory operations that need to take place (like moving states between emulator and device). This may be a basic feature, but it comes in handy when calls take a long time to complete and block the main thread. Examples are when Avatar is waiting for a breakpoint to be reached, or a significant memory copy is taking place over a low bandwidth connection. Plugins can also be loaded into S2E from the configuration file to trace execution paths taken by QEMU.

[1] Authors note: I spent several months debugging to find that Thumb instructions are decoded differently than ARM instructions, and use different instruction address schemes. Reviewing this file early on would have indicated that clearly and potentially saved a significant amount of time.

4.2 Key Concepts and Methodology

To make automatic exploit generation on embedded systems possible, all the previously discussed techniques, technologies and frameworks must be harnessed in such a way that cooperation is achieved between all components. This section explains the overall methodology that is followed through the implementation of automatic exploit generation, in respect to all the techniques utilised.

The researcher first needs to obtain a copy of the firmware under analysis. This can be achieved in two possible ways, with the conventional methods being extracting firmware from the physical device, or downloading a copy of the firmware. Extracting the firmware directly from the physical device is useful since the firmware under analysis will always be the exact version that is running on the device, meaning no unexpected issues can arise. Extracting firmware can be achieved by connecting to the target device with a debugger, such as OpenOCD, and dumping the contents of the code flash region to a file. Downloading firmware from the internet is also a viable option. Typically, firmware is distributed in packed update files, which must be extracted to yield the firmware bytecode. This must then be flashed to the target device to ensure that during analysis, the emulator and target device are both running the same code. Of course, both methods assume that the developer is working with a target device which has not had debugging functionality permanently disabled. This is a common protection applied to consumer hardware to prevent misuse and to protect intellectual property. For a researcher or developer working in a legitimate environment, this will never be a problem.

Next, the firmware must be disassembled from machine code into assembly, using a tool like objdump. This is necessary as Avatar will be utilising context switching functionality, to enable the target device to execute the firmware natively, to complete all hardware initialisation procedures before control is handed over to the emulator. Once the firmware has been disassembled, an instruction address needs to be selected for the breakpoint to be placed after all hardware initialisation has completed. Generally, all hardware initialisation is completed in the first few function calls of the main procedure, and it should be clear when they have all been completed.

Avatar can then be initialised, starting an emulator for the specific processor model under analysis. The previously obtained firmware file is then loaded to the correct memory range in the emulated processor. A connection to the target device is then created using OpenOCD, and the target halted. The firmware should be reflashed to the device on each analysis attempt, to ensure integrity of the firmware under test. After reflashing has been completed, a breakpoint is set to the previously identified address which marks the completion of all hardware initialisation. The target device is then resumed, and can execute until the breakpoint is reached.

At this stage all hardware initialisation has been completed on the target device, avoiding any issues of critical timing or hardware polling loops which would have been inconvenient to complete on the emulator. It is time for the context switch to the emulator. The entire state of the target device needs to be copied to the emulator through the debugging channel. All memory addresses which address to RAM need to be copied, and every register needs to be transferred. Note that memory is typically transferred one word at a time, which can make this operation time consuming for microprocessors with a large amount of addressable RAM. This is due to Avatar querying OpenOCD for the contents of each memory address individually, and then forwarding each result to QEMU separately through a GDB server. A more elegant solution which can copy larger chunks of memory at a time and reduce the number of packets transmitted should be investigated for future projects.

The emulator can then begin execution of the firmware from where the target device was halted by the breakpoint. All memory read and write requests are forwarded to Avatar, and if necessary, Avatar will forward the request to the target device for memory-mapped IO communication. Otherwise, state is maintained inside the emulator. S2E marks memory regions where symbolic execution should be active, and sets breakpoints for calls to Lua callback functions to occur for symbolic memory manipulation.

While all of has been happening, QEMU has been translating all ARM / Thumb instructions to TCG instructions and executing them. When the program counter reaches a region marked for symbolic execution, S2E invokes KLEE. KLEE then takes the current TCG instructions and translates them to LLVM instructions. From there, the symbolic execution engine will call Lua callback functions to mark specific memory variables as symbolic. KLEE then spawns a symbolic interpreter, and symbolic execution takes place over the firmware.

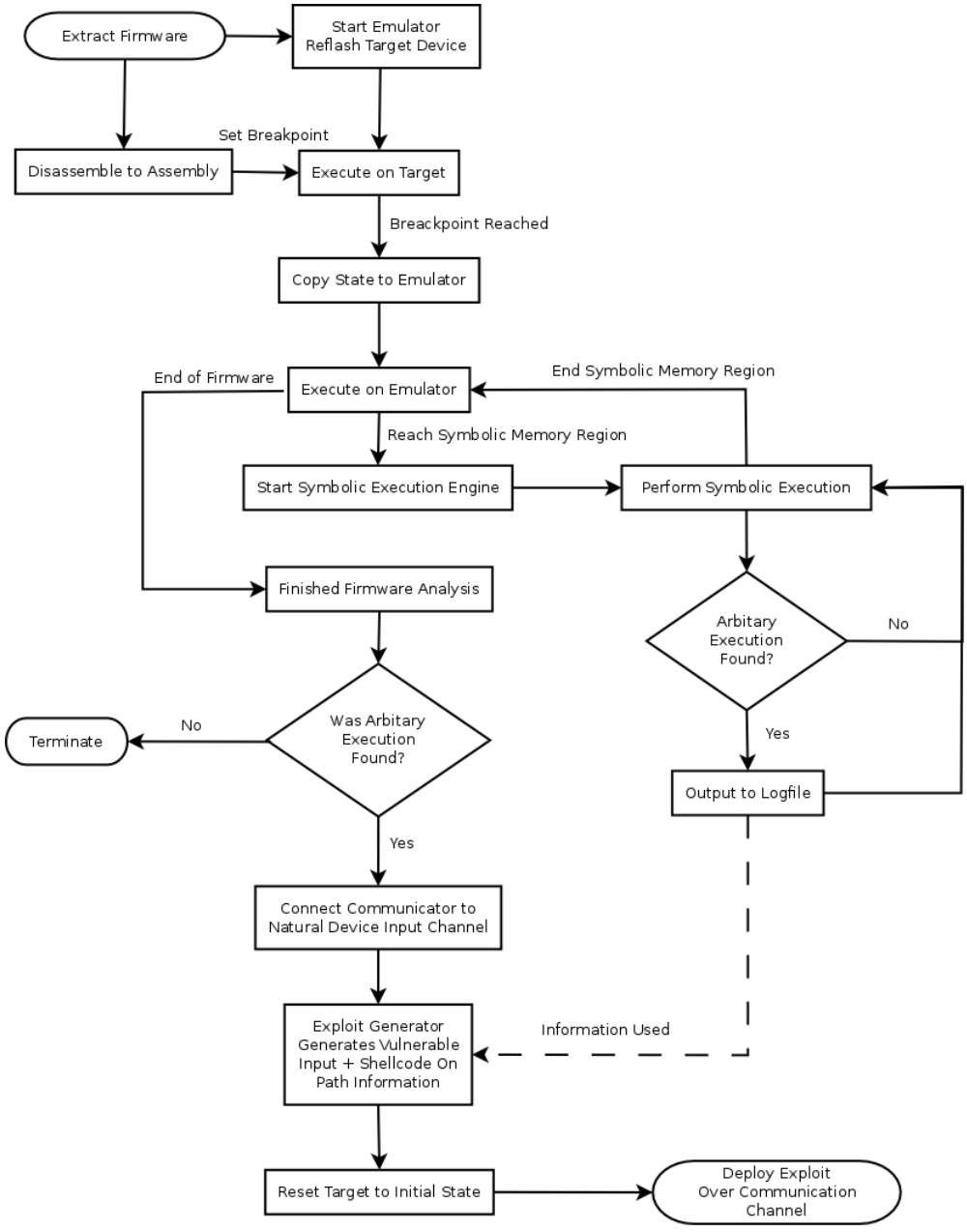

Figure 4.2: Methodology behind automatic exploit execution on embedded systems

As the symbolic interpreters execute the firmware and fork more interpreters upon control flow conditions, each interpreter is building a logical formula of path conditions. Each interpreter is looking for arbitrary execution conditions, as defined by the ArbitaryExecution S2E plugin. The most common condition will be when a symbolic variable is copied into the program counter on termination of a function. In this scenario, the interpreter is halted and the STP SMT solver is queried to find concrete values for symbolic variables that satisfy the path formula. These values are effectively the inputs required to place the firmware into a vulnerable state. These concrete values are then written to logfiles and output in the console.

Of course, arbitrary execution conditions may not be found on the first round of symbolic execution, if many interesting memory regions are identified. In this case, KLEE will terminate all symbolic states when the symbolic marked memory region is exited, and will stop translating TCG instructions to LLVM. Execution continues normally in QEMU, and control can return again to KLEE when the program counter enters the next symbolic memory region.

Eventually, the end of the firmware will be reached. If the firmware utilises an infinite main loop, a Lua callback function will need to be set on the final jump instruction to terminate QEMU and KLEE. At this stage, all firmware analysis is complete. The emulator is shutdown and the target device is halted. Any breakpoints that still remain on the target device need to be removed. This marks the end of Zaddach *et al.* Avatar framework, and the beginning of the extensions to Avatar, which are the real contributions of this research project.

To be able to successfully exploit the target device, there needs to be a communication mechanism with which the exploit can be delivered to the target device from Avatar. Since embedded systems can accept input from a variety of sources, this mechanism needs to present a generic interface that can be extended to suit any specific input method. This research project introduces the Communicator class, which presents a generic interface for channel initialisation, connection and disconnection, and reading and writing of data. The researcher can then simply extend the Communicator class and implement a specific communication interface for the channel used in their target device.

Next is the task of generating an exploit for the firmware. There are many classes of exploit that can be automatically generated, so much like the Communicator class, this research project introduces the ExploitGenerator class. The ExploitGenerator class presents a generic interface which can be extended to suit any exploitation technique. ExploitGenerator treats exploits as the concatenation of two elements: the inputs necessary to place the device in a vulnerable state, and the shellcode to be executed. The researcher can simply extend ExploitGenerator to construct input information from concrete path information gathered from the ArbitaryExection plugin, and specify how shellcode is generated or load generic shellcode from a file. ExploitGenerator also contains an instance of the Communicator class to provide a single interface to deploy exploits to the target device.

Once analysis of the firmware has been completed and the emulator has been terminated, the researcher can use a specific implementation of the Communicator class to connect to the target device over its natural communication channel. A specific exploit generator can then created and passed a reference to the Communicator object. Vulnerable inputs and shellcode can then be generated, which are then concatenated to produce an exploit.

The device is then prepared for exploitation by being reset to its initial state. To be able to verify that the exploit functions as expected, a breakpoint needs to be set on the instruction that triggers the exploit. In the case of stack buffer overflow exploits, this is the call to pop the return address from the stack into the program counter. Once this breakpoint has been set, execution can be resumed on the device. The ExploitGenerator can then deploy the payload to the device over the natural communication channel, and the exploit should be triggered. This should cause the breakpoint to halt the device, and the researcher can then examine the registers using OpenOCD to verify that the program counter has been hijacked. Execution of the payload can be single stepped with the debugger to ensure that everything functions as required.

4.3 Implementation

4.3.1 Avatar Configuration

The Avatar configuration file is the core Python script that controls the operation of the Avatar framework. This file imports all relevant libraries for analysis, and contains configuration parameters required for S2E to function, along with all the analysis logic. The configuration file is the Python script to be executed to facilitate analysis of the target embedded system. New configuration files must be created for each individual firmware, as analysis is tailored to each executable.

S2E requires considerable configuration in order to operate. Firstly, the hardware of the target device needs to be specified in order to create a virtual machine that closely emulates the target processor. QEMU has definitions of many ARM processor families, in which one must be selected to be the target processor. Memory ranges 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. Of course, if code and RAM are not mapped, then all memory operations will be forwarded to the target device, resulting in the previously mentioned full-seperation mode.

Plugins that are loaded directly into S2E must also be configured. The most notable include the RawMonitor, ModuleExecutionDetector and Annotation plugins. 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. The Annotation plugin allows the researcher 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. Roughly speaking, each every state of the flowchart in Figure 4.2 in the Key Ideas and Methodology section represents one or a small group of function calls in the Avatar configuration file. Since the configuration file is written in Python, analysis is very procedural, which further maps the function calls required in Avatar to a structure found in the flowchart of Figure 4.2.

Typically, there is no need for concurrent actions in the Avatar framework, as procedural mechanisms will generally work with any firmware analysis. The only exception comes when automatic exploit generation is required. Due to timing issues, the device needs to be reset concurrently while the natural input communication channel is being constructed and an exploit generated. This is to ensure that the device is natively executing and is ready for exploitation when the payload is sent down the communication channel. A simple pair of threads can easily accomplish this task, with one thread resetting the device and setting breakpoints to help verify exploitation, while the other is setting up communication channels and generating the exploit.

4.3.2 Natural Device Input Communication

All embedded systems read input from some natural input channel to be useful. The problem is, most frameworks (including Avatar) have no way to communicate with the target device over its natural communication channels. If input is ever needed to be injected into the target device, a debugger is typically used to modify the contents of received data to the injected data.

This is bad for numerous reasons, with the main issue being 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 natural communication channels. That is, injecting data assures integrity, while in the real world we can never assure integrity. Take a UART serial port for an example. The data to be injected to the firmware could contain machine code that could be interpreted as ASCII code for newline or carriage return characters. Injecting the data into the firmware via a debugger will assure integrity, and all bytes will be loaded into the firmware exactly as contained in the data. However, if this data was to be sent over a UART serial channel, the UART transmitter software or the physical device would interpret the bytes that map to ASCII carriage return characters as bits that designate the end of transmission. This would cause only parts of the firmware to be copied, meaning integrity is not assured.

For this reason, it is important to send exploit payloads down the natural communication channel that would be used in real world exploitation scenarios. This research project introduces an extension to the Avatar framework which enables the researcher to do exactly that. The Communicator module[2] presents a generic interface of abstract functions for implementing channel initialisation, connection, disconnection, reading and writing. The researcher 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. 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.

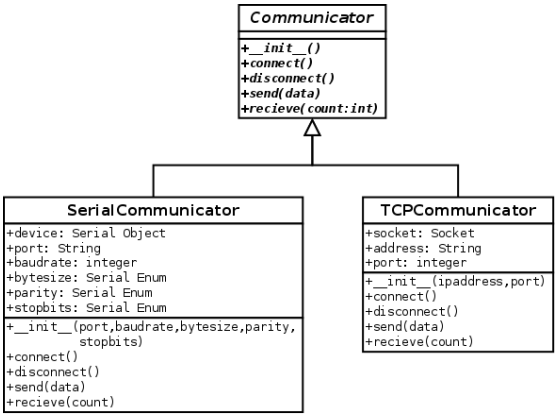

Figure 4.3: UML depicting the Communicator module

To illustrate the use of the Communicator module, a concrete implementation of a serial UART device communicator and TCP socket communicator[3] have been provided. The serial UART implementation depends on the Python library PySerial[4], which implements serial port communication for TTY devices. The researcher can specify the device address, baud rate, variable bytesize, bit parity and the number of stop bits. Since the communicator modules is an extension of Avatar, all centralised logging facilities are supported and both communicators can log exceptions to the Avatar logfile.

[2]The Communicator class is available in the "communicators" folder in the Avatar directory. See Appendix.

[3]The serial and tcp communicator classes are available in the "communicators" folder in the Avatar directory. See Appendix.

[4]http://pyserial.sourceforge.net/

4.3.3 Exploit Generation

The ExploitGenerator module[5] is the extension to the Avatar framework which facilitates automatic exploit generation. Since there are various exploit classes, the ExploitGenerator class presents a generic interface which can be extended to suit any exploitation method, such as stack buffer overflows, return oriented programming, use after free and null pointer dereference.

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, the 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 researcher also has the option to manually override the automatically generated input and payload variables if they so choose.

To deploy the exploit to the target device, the ExploitGenerator class sends the exploit down a previously created natural 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.

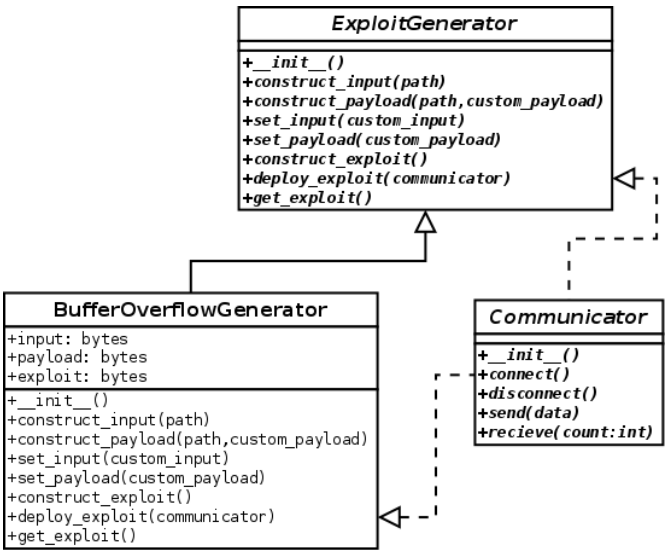

Figure 4.4: UML depicting the ExploitGenerator module

The goal of this research project is to automatically generate exploits for stack buffer overflow vulnerabilities. BufferOverflowGenerator[6] is a concrete implementation of ExploitGenerator which implements this feature. 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 then concatenates the input and payload to generate an exploit and deploys to the target device through a specified communication channel.

[5]The ExploitGenerator class is available in the "exploitgenerators" folder in the Avatar directory. See Appendix.
[6]The BufferOverflowGenerator class is available in the "exploitgenerators" folder in the Avatar directory. See Appendix.

5 Evaluation

5.1 Equipment Used

The embedded system used in the development and evaluation of this research project 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.

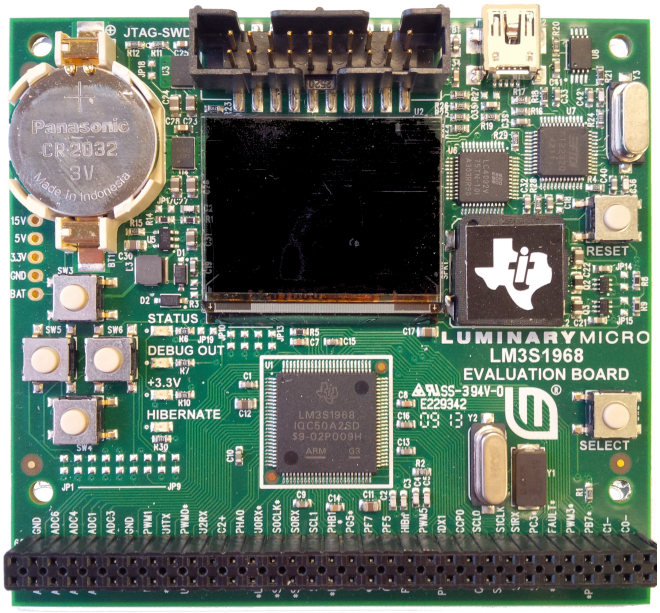

Figure 5.1: The Stellaris EKS-LM3S1968 Evaluation Kit (cables omitted)

The LM3S1968 microprocessor belongs to the ARM Cortex-M3 family, which was specially designed for low power embedded systems. Cortex-M3 processors implement ARMv7-M architecture, and supports the Thumb and Thumb2 instruction sets, which utilise both 16 and 32-bit length instructions. Note that the traditional 32-bit ARM instruction set is not supported. The Stellaris board contains multiple specialised hardware peripherals which are managed through memory-mapped IO operations. The most obvious is the OLED display with a resolution of 128x96 pixels, each configurable to 16 levels of grayscale. There are 52 GPIO pins which are used for tasks such as interfacing 6 push buttons, or the magnetic speaker. The board also implements 3 fully programmable UART serial channels, 8 10-bit Analogue-to-Digital Converters (ADC), 3 Pulse-Width Modulation (PWM) generator blocks and full USB and Ethernet support. Texas Instruments also provide an excellent development environment and device driver library, which makes this board ideal for the purposes of this research project.

Natural device communication between the Stellaris board and the Avatar framework was achieved over a serial UART line. Serial UART lines are a common feature in real world embedded systems, which makes this communication mechanism the logical choice. 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 which supports the CP2102 UART chip.

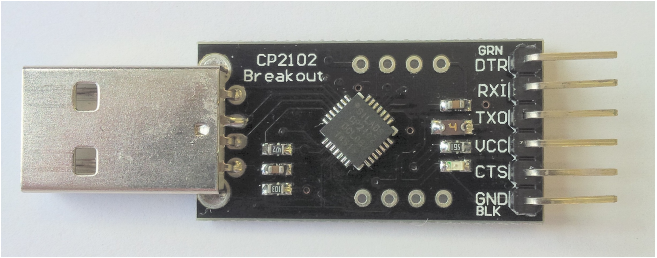

Figure 5.2: The CP2102 External USB UART TTY adapter

Male to Female jumper leads were used to connect the two devices together. The hardware was configured as follows: The RXI and TXO pins (receive and transmission lines) on the USB UART were connected to the U0TX and U0RX ports on the Stellaris board respectively. The GND (ground) pin was connected to a free GND port on the Stellaris board. The CTR and CTS signalling pins were unused, as the firmwares implemented their own mechanisms for determining if a message has been received. VCC was not needed since both devices ran off the same voltage supplied from the host computer's USB ports (3.3v). The serial channel was configured to 38400 baud, with eight bits per byte, one stop bit and no parity bits. Figure 5.3 shows the completed connections.

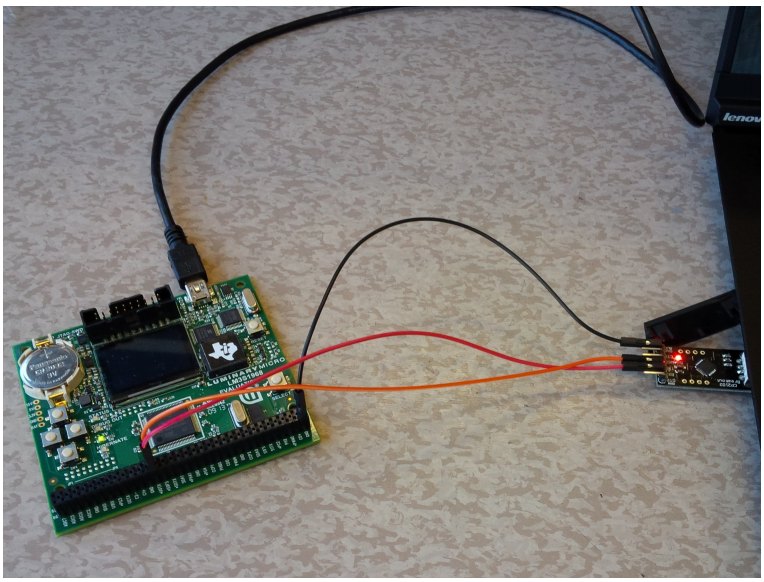

Figure 5.3: The Stellaris board connected to the external USB UART TTY

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. See the Appendix further information on software packages that were installed, and custom packages that were required to be compiled.

5.2 Vulnerable Firmwares

Three vulnerable firmwares were developed specifically to evaluate the implementation and key ideas of this research project. Developing our own firmwares enabled us to limit the scope and complexity of each firmware down to an acceptable level, in order to keep the run times of symbolic execution manageable. It also provides insight into what operations take place inside a firmware, which greatly helps to find appropriate sections for symbolic execution and where annotations should be placed. Finding those locations requires manual disassembly of the firmware and careful reading of the assembly code, which gets several orders of magnitude more difficult when analysing an unknown firmware downloaded from the internet. Searching the internet for other firmwares for the Stellaris board yield very few results, which would also make comparisons between the firmwares difficult since they achieve different tasks with different hardware peripherals. Developing our own firmwares enables us to be consistent in firmware design, to create comparable firmwares of increasing complexity.

Each of the firmwares implemented 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 that peripheral by any shellcode executed. Further, use of specialised hardware peripherals allows the firmwares developed to showcase the features of the Avatar framework, and show that this research project achieves the goals outlined in the Introduction.

The firmwares developed share a common intentional vulnerability that is exploitable on some or all program paths of the firmwares execution. The vulncpy() function, discussed at length in the Background section, introduces a simple stack buffer overflow vulnerability since it does not perform any length checking of an array passed as a parameter. Vulncpy() is called after the firmwares receive a message over the serial UART line, which contains tainted data which is entirely attacker controlled. To evaluate this project, all three firmwares will be analysed by the extended Avatar framework, and if successful, the vulnerability should be detected through symbolic execution, and an exploit generated from provided vulnerable path information.

As mentioned previously, manually disassembling firmwares to place annotations is one of the more difficult steps required for analysis. Since each firmware shares the same core vulnerability, the annotation is effectively the same for each firmware to mark the message buffer of tainted data as symbolic. Consider an excerpt of the *Small* firmware:

```
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
```

Annotations need to be placed at sections of the firmware where variables or buffers 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 simply 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. Marking the buffer as symbolic takes slightly more work.

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
```

Figure 5.4: Annotation callback function marking buffer as symbolic

This 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 sharing the same vulnerability and utilising similar annotations, each firmware is comparable with how vulnerabilities are found and exploited.

The three firmwares developed are designated as *Small*, *Medium* and *Large*. Small and Medium share a similar codebase, with Medium being slightly more complex with more control flow structures. Large is an attempt to provide an in-vivo example of a complicated real world firmware, and is of several orders of magnitude more complicated.

5.2.1 Small

Small is around 30 lines of code, and is very simplistic. 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.

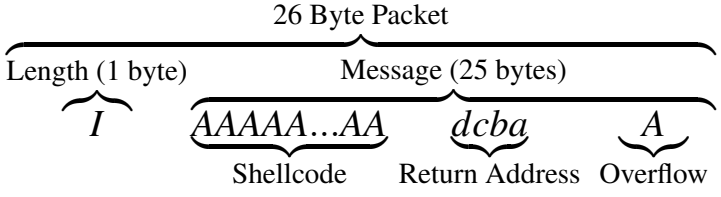

Figure 5.5: Small packet structure and example exploit

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 shellcode presented in this example exploit consists of 20 bytes to fill the buffer in vulncpy(), 4 bytes to overwrite the return address to the desired value `0xabcd` (note that ARM Cortex-M3 is little endian), and a further byte which overwrites the previous stack frame.

5.2.2 Medium

Medium revolves around the same ideas used in Small. Medium initialises hardware peripherals, receives a `length` and two magic bytes along with a message over a serial UART line. Once the message is received, if the magic bytes fit within certain restrictions, the message is directly passed to vulncpy(). This additional control flow logic was introduced to prove that symbolic execution really can find all paths through the program, and generate the inputs required to place the firmware into a vulnerable state.

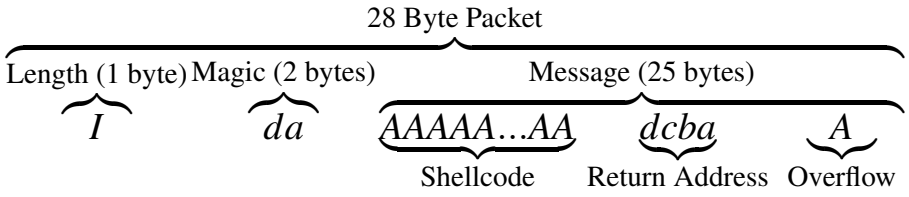

Figure 5.6: Medium packet structure and example exploit

With the addition of the magic bytes, the inputs required to place Medium into a vulnerable state include a `length` value which is greater than 20, the first magic byte must be exactly equal to the ASCII 'd' character, and the final magic byte must be less than the ASCII 'e' character [1]. The shellcode takes exactly the same form as used above in Small, since it passes through the same vulncpy() function.

5.2.3 Large

Large is an attempt to recreate an in-vivo example of a real world firmware, and is over 600 lines long[2]. Large implements 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 first, is a menu screen which displays a list of available options. The user can then press one of the directional buttons on the Stellaris board, or send a packet to the device over UART with a command to change the view. The first view, denoted by the up button or command 1, parses and pretty prints an attached message to the screen. The second view (command 2), displays an about screen. The third view (command 3) displays a help screen. The fourth view (command 4) allows the user to send a message. Messages are constructed by pressing the up and down buttons to change characters and left and right to move to the next character.

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 goal of Large is to see if the extensions to Avatar can generate exploits for a complex application with problematic components.

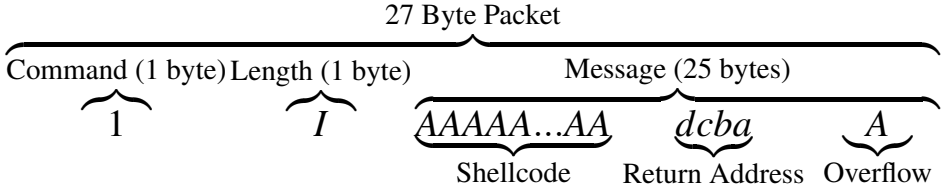

Figure 5.7: Large packet structure and example exploit

Large can be placed into a vulnerable state by sending command 1: receive and display message. The message is parsed and vulncpy() is called before the message is printed to the screen. The shellcode follows the same format used in the two previous firmwares.

[1] The source code for Medium can be found in medium.c in avatar-stellaris/medium/firmware. See Appendix.
[2] The source code for Large can be found in large.c in avatar-stellaris/large/firmware. See Appendix.

5.3 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 reads `length` in as a printable ASCII character, the `length` is offset by the character '0'[3], or `0x30`. This means that the SMT solver was tasked to find values greater than 20 which include the offset. Two exploits are shown below in Figure 5.8:

```
Length  Shellcode                                          Return    Overflow
4B      6161616161616161616161616161616161616161616161    b8000020  61
7F      6161616161616161616161616161616161616161616161    b8000020  61
```

Figure 5.8: Exploits generated for Small, shown in hexadecimal form. Spaces added for clarity

In both of the above 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. The return address is set to the start of the buffer, and since the Stellaris board is a simple embedded system with no sophisticated memory protection features, 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.

Medium builds upon the same ideas that Small presents, and adds two extra magic bytes to increase control flow logic that must be traversed in order to generate a successful exploit. In Figure 5.9, `M1` and `M2` represent those magic bytes, with `M1` allowing exploitation only if `M1` is exactly equal to 'd' (`0x64`), and `M2` being less than 'e' (`0x65`). The length and shellcode bytes follow the same conditions as Small.

```
Length  M1  M2  Shellcode                                          Return    Over
49      64  64  6161616161616161616161616161616161616161616161    b0000020  61
5E      64  00  6161616161616161616161616161616161616161616161    b0000020  61
73      64  50  6161616161616161616161616161616161616161616161    b0000020  61
```

Figure 5.9: Exploits generated for Medium, shown in hexadecimal form. Spaces added for clarity

The exploits generated consistently set the first magic byte to the correct value of `0x64`, and the second magic byte to below `0x65`, meaning that through symbolic execution and the SMT constraint solving process, correct values are gained. Again, the return address is the same on each exploit, since the buffer is always allocated in the same place. By debugging Medium, the buffer was found to be allocated between `0x200000B0` and `0x200000C9`, which agrees with the generated exploits.

Difficulties were encountered in the analysis of Large, which prevented any exploits from being generated. Firstly, Large contains a significant amount of execution paths, and since each path has multiple entry points from different input mechanisms (such as push buttons, UART packets), symbolic execution would fork states to explore already explored paths. This lead to severely degraded performance and vulnerable states were not being discovered. Secondly, the firmware is structured such that all tasks are scheduled in an infinite loop. Instead of terminating at the end of analysis, the emulator would simply complete another loop of the firmware, entering the symbolic execution marked region over and over. No additional information was being gained each loop, and the only way to terminate the emulator was to terminate the parent Python script performing the analysis, which prevented any exploits from being automatically generated. Lastly, it appears that utilising the OLED display requires a time critical operation which could not be accommodated for once execution had been moved to the emulator as a part of the original context switch.

Hence, automatically generating exploits for (simple) embedded systems firmwares is a success, although it is regrettable that difficulties were encountered with complex in-vivo firmwares.

[3] Best explained by examining the source code in avatar-stellaris/small/firmware/small.c. See Appendix.

6 Discussion

This chapter highlights various limitations produced by this research project, and the impacts and restrictions those limitations have placed on the implementation of automatic exploit generation for embedded systems. Future work is discussed, suggesting research topics to further increase the power and usefulness of automatic exploit generation, and how the current limitations could potentially be addressed.

6.1 Limitations

The largest limitation of the current state of implementation is that customised shellcode cannot be generated for each individual firmware. This does not necessarily have a large impact on this research project, since the definition of an exploit is the concatenation of inputs that place the device into a vulnerable state, with shellcode which exploits that vulnerable state. The shellcode must still be produced somehow, and this project assumes that the researcher is capable of producing it. In reality, a developer would not be able to. Generating shellcode is not a simple process because buffers can be of different lengths, and shellcode either needs to fit into small buffers, or be padded out to fill larger ones. At the same time, every firmware is different, and hardware peripheral driver modules are nearly always linked to the firmware in different locations. This means that if shellcode wishes to hijack execution and output words to the display, Avatar / ExploitGenerator module would need to scan the firmware to find what function to call, and what address it resides at. If a machine code firmware with no debug symbols is present, this becomes an extremely difficult problem.

While this research project relieves developers of the task of searching for vulnerabilities and generating exploits themselves, they still need to be able to set up and configure Avatar. While the previous chapters show that this is a straightforward task, it is still difficult. On average, the Avatar configuration files for the three vulnerable firmwares are around 450 lines of code. This may seem unwieldy if the firmware under analysis is 30 lines of code, but the redeeming factor is that for similar firmwares running on the same target device, there is very little modification that needs to take place to port a configuration file from one firmware to another. This can be seen in practice, with the configuration files of Small being largely the same as Large, with the major changes happening around where annotations are called. For developers working in practice, the configuration file needs to be setup once, and then can be used many times afterwards. The initial setup can be performed by an expert, leaving developers to run and interpret the analysis.

Avatar is not scalable for large complex embedded systems. The action of performing symbolic execution and passing every memory-mapped IO peripheral access over USB to the target device is too slow for real world analysis. Consider a baseband processor present in cellular phones. Baseband firmwares are typically several megabytes in size, and utilise many nested loops and state machines to implement the GSM protocol. Symbolic execution would likely tend towards state space explosion, time critical radio peripheral accesses will likely fail, and the USB debug channel will probably not have enough bandwidth to cope with the interrupts generated, alongside the legitimate forwarded memory-mapped IO peripheral accesses.

Due to the nature of Avatar, the researcher must have a physical target device present for analysis. While this is not an issue in most scenarios, a problem is presented when the target device does not exist and is in development. Typically, developers will have access to an emulator of the target device to implement firmware, which is not sufficient for Avatar to complete an accurate analysis in the development timeframe.

6.2 Future Work

It would be interesting to see what benefits could be obtained by implementing dynamic taint analysis in S2E. This would remove the need for annotations entirely as variables and buffers could simply be marked symbolic if they originated from a taint source. This would decrease the overall difficulty of setting up the Avatar framework, as only taint sources would need to be marked, not individual variables and buffers. If quantitative dynamic taint analysis is implemented, then a new path selection algorithm can also be implemented, which would select paths that contain more tainted data variables than other paths. This could potentially narrow the search scope down faster than previously discussed algorithms, and speed up the symbolic execution process.

Currently, vulnerability detection is overly general, and 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. Research is required 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 the researcher needs 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.

Solutions need to be found to increase the performance of Avatar for large, complex firmwares. As higher processing power becomes cheaper with newer chips manufactured, firmwares will become more complex over time. While automatic vulnerability analysis and exploit generation is far more scalable than traditional manual analysis, design and architecture changes need to happen early in the implementation process. While most of the performance issues are contained within S2E, communication over USB is still an issue. Since embedded systems are designed to be as cost effective as possible, fast communication channels such as USB 3.1 or Thunderbolt will not be a common feature on most embedded systems. Therefore, the protocols and information over those protocols need to be tuned for performance, which potentially means moving away from current human readable JSON packets to a machine protocol.

The most valuable and urgent feature would be to implement shellcode generation for every ExploitGenerator. Generating shellcode requires extensive analysis of each individual firmware to find the previously mentioned buffers, their lengths, and addresses of useful functions that can be utilised to perform an action during a control flow hijack. Shellcode also requires knowledge of the runtime environment, such as the size of stack frames, and the addresses of allocated buffers. The real issue lies within the fact that every class of vulnerability is exploited in different ways, and simply collecting the information required for one exploit may not be enough for another exploit. The best solution would be to find what are the most common elements used for shellcode and extend Avatar to provide easy access to that information.

On a practical note, the extensions to Avatar need to be tested with new, real world scenarios out of the scope of this research project. Such scenarios could include analysing the firmware running on a modern smartwatch. New natural device Communicators would need to be implemented for Avatar to communicate over wireless channels such as Bluetooth, and ExploitGenerators would need to be implemented for more sophisticated vulnerability classes, such as ROP. This would test the extensibility of the additions implemented in this research project, and how flexible Avatar is as a whole.

Lastly, this project is open for anyone to work on. The code used in this research project has been uploaded to Github [1], and anyone can begin working with and extending the project themselves. Automatic exploit generation is a new and exciting field of computer science, and there are many more possibilities than those mentioned here. The hardest part is simply getting started.

[1] Both the extended Avatar framework and firmwares used in Evaluation with configuration files have been uploaded. See Appendix.

7 Conclusion

The aim of this research project was to extend an existing dynamic analysis framework to implement automatic exploit generation for embedded systems firmwares. By automating vulnerability discovery and exploit generation, firmware developers can search for and generate proof of concept exploits for vulnerabilities during the development phrase of an embedded systems lifecycle. This means that software flaws can be fixed early on, and that developers can complete quick security audits without having to possess an expert understanding of low level hardware and exploitation techniques.

This report presented an in-depth review of a common exploitation method, stack buffer overflows, and two different techniques used to detect vulnerabilities, dynamic taint analysis and symbolic execution. Stack buffer overflow vulnerabilities occur when a copy operation copies more data into a buffer than was previously allocated, and overwrites the return address, which can be modified to execute attacker provided code. Dynamic taint analysis is the process of monitoring tainted data as it flows through an application, and can present warnings when tainted data could hijack execution. Symbolic execution dynamically executes a firmware using interpreters which fork upon changes in control flow. Each interpreter builds a logical formula of execution, which can be evaluated when a memory unsafe action occurs to determine what inputs can satisfy the potential vulnerability.

The Avatar dynamic analysis framework by Zaddach *et al.* was comprehensively examined and all functionality reviewed. Avatar mediates communication between a target embedded system and an emulator which emulates that system. The firmware under analysis is typically executed inside the emulator, and forwards all memory-mapped IO accesses to the target device for reading correct responses from specialised hardware peripherals. The emulator supports symbolic execution to find memory unsafe states, which presented an ideal foundation for this project. This project implemented extensions to Avatar to provide natural device communication over expected communication channels, and added exploit generation modules which generate exploits based on vulnerable path information provided through symbolic execution. An analysis flow was devised to perform dynamic analysis over firmwares, by executing on the target device until hardware was initialised, and then moving execution to the emulator to perform symbolic execution. Symbolic execution was used to find inputs which place the device into a vulnerable state, which was then used to generate exploits via the implemented ExploitGenerator. The exploit is then delivered to the target device through a natural device Communicator, which then triggers the vulnerability. Developers can use this to fix vulnerabilities and verify that the fix has worked, with subsequent runs of the analysis framework.

Three intentionally vulnerable firmwares for the Stellaris EKS-LM3S1968 Evaluation kit were developed specifically for this project to evaluate the solution. Each of the firmwares share the same vulnerability, but differ in complexity, in order to evaluate the usefulness of the extensions added to Avatar. Small reads in data over a serial UART line, and was simple in nature and quickly exploited. Medium provided slightly more complexity than Small in terms of control flow statements, and was also exploited. Large attempted to provide an in-vivo example of a real world firmware and was several orders of magnitude more complex than Small and Medium. Large implemented a feature rich message passing application which can send and display messages received over a serial UART line. Difficulties were encountered in exploiting Large, meaning that automatic exploit generation for embedded systems firmwares is indeed possible, but not yet practical for complex firmwares which require time critical access to specialised hardware peripherals.

Bibliography

[1] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. Aeg: Automatic exploit generation. In *NDSS*, volume 11, pages 59–66, 2011.

[2] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. Automatic exploit generation. *Communications of the ACM*, 57(2):74–84, 2014.

[3] 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.

[4] Oleksandr Bazhaniuk, John Loucaides, Lee Rosenbaum, Mark R Tuttle, and Vincent Zimmer. Symbolic execution for bios security1. 2015.

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

[6] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 143–157. IEEE, 2008.

[7] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[8] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 380–394. IEEE, 2012.

[9] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. *S2E: A platform for in-vivo multi-path analysis of software systems*, volume 39. ACM, 2011.

[10] Andrei Costin, Jonas Zaddach, Aurélien Francillon, Davide Balzarotti, and Sophia Antipolis. A large-scale analysis of the security of embedded firmwares. In *USENIX Security Symposium*, 2014.

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

[12] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *USENIX Security*, pages 463–478, 2013.

[13] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[14] Josselin Feist, Laurent Mounier, and Marie-Laure Potet. Statically detecting use after free on binary code. *Journal of Computer Virology and Hacking Techniques*, 10(3):211–217, 2014.

[15] Aurélien Francillon, Daniele Perito, and Claude Castelluccia. Defending embedded systems against control flow attacks. In *Proceedings of the first ACM workshop on Secure execution of untrusted code*, pages 19–26. ACM, 2009.

[16] Vijay Ganesh and David L Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification*, pages 519–531. Springer, 2007.

[17] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 474–484. IEEE, 2009.

[18] Shih-Kun Huang, Min-Hsiang Huang, Po-Yen Huang, Chung-Wei Lai, Han-Lin Lu, and Wai-Meng Leong. 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*, pages 78–87. IEEE, 2012.

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

[20] Collin Mulliner, Nico Golde, and Jean-Pierre Seifert. Sms of death: From analyzing to attacking mobile phones on a large scale. In *USENIX Security Symposium*, 2011.

[21] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[22] Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

[23] Jonathan Pincus and Brandon Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *Security & Privacy, IEEE*, 2(4):20–27, 2004.

[24] Marco Prandini and Marco Ramilli. Return-oriented programming. *Security & Privacy, IEEE*, 10(6): 84–87, 2012.

[25] Dominic Rath. Open on-chip debugger, 2008.

[26] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 317–331. IEEE, 2010.

[27] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *USENIX Security Symposium*, 2011.

[28] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 48–62. IEEE, 2013.

[29] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 497–512. IEEE, 2010.

[30] Ralf-Philipp Weinmann. Baseband attacks: Remote exploitation of memory corruptions in cellular protocol stacks. In *WOOT*, pages 12–21, 2012.

[31] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. 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.

A Appendix

A.1 Overview

Implementing automatic exploit generation for embedded systems is a complex task and requires a significant amount of software to achieve this task. Things are further complicated when most software components are required to communicate with other components. This section serves as a guide on how to successfully build and run the necessary software to recreate this research project. A significant amount of time of this research project was sunk into installing and configuring software alone, and issues frequently arose. Be prepared to also face issues, which is why it is recommended that only people with an advanced or elite knowledge of Linux and software compilation attempt this.

This section is loosely based on the instructions given by Zaddach *et al.* on the Avatar website:
`http://s3.eurecom.fr/tools/avatar/`
All code developed for this project can be accessed on Github: `https://github.com/msr50`

A.2 Installing Software

Components that are required for this research project include: the extended Avatar Framework, Avatar configuration and vulnerable firmwares used in Evaluation, S2E, OpenOCD and an ARM toolchain.

It is recommended that a 64 bit Debian Wheezy virtual machine is used for this activity. Specifically, Debian 7.8. While it may be possible to get things working on other distributions such as Fedora, frequent issues will arise with new compilers being incompatible with the large and fragile codebases used in this project.

The following directory structure will be used:

```
# Create the necessary directory structures
mkdir ~/Workspace
mkdir ~/Workspace/COSC460
```

Next we install the extended Avatar framework worked on for this research project.

```
# Install extended Avatar Framework
# Install Python and PIP
sudo apt-get install python3 python3-pip

# Install Avatar from github
sudo pip-3.2 install git+https://github.com/msr50/avatar-python.git#egg=avatar

# We also clone a local copy for ease of reading
cd ~/Workspace/COSC460
git clone https://github.com/msr50/avatar-python.git

# The added extensions are available in avatar-python/avatar/communicators/*
# and avatar-python/avatar/exploitgenerators/*
```

The Avatar configuration and vulnerable firmwares that were used to evaluate this research project can be obtained by:

```
# Change to the correct directory
cd ~/Workspace/COSC460

# Fetching the avatar-stellaris vulnerable firmwares
git clone https://github.com/msr50/avatar-stellaris.git

# All files can be found in their respective folders
```

The next essential component is to install S2E. These instructions are provided by Zaddach *et al.*

```
# Move to the correct directory
cd ~/Workspace/COSC460

# Install all build-dependencies
sudo apt-get build-dep qemu llvm
sudo apt-get install build-essential flex subversion git gettext liblua5.1-dev \
  libsdl1.2-dev libsigc++-2.0-dev binutils-dev python-docutils python-pygments nasm bison

# Get the source code from github
git clone https://github.com/eurecom-s3/s2e.git

# Make it building out-of-tree
mkdir build
cd build
make -f ../s2e/Makefile

# This will take some time to build...
```

OpenOCD, the device debugger needs to be compiled from source, since we require support for the FTDI USB debugging chip present on the Stellaris board. Use the latest version available. At the time of writing, 0.9.0.

```
# Move to the correct directory
cd ~/Workspace/COSC460/

# Fetch the OpenOCD source distrubution
wget http://sourceforge.net/projects/openocd/files/openocd/0.9.0/openocd-0.9.0.tar.bz2

# Uncompress archive
bunzip2 openocd-0.9.0.tar.bz2
tar -xf openocd-0.9.0.tar

# We need to install the FTDI chip driver, so install libusb
sudo apt-get install libusb-1.0-0*

# We also need to get any additional dependancies
sudo apt-get build-dep openocd

# Build openocd
cd openocd-0.9.0/
# Ensure that FTDI support is enabled. Should see a FTDI [OK] in output.
./configure --enable-ft2232_libftdil --enable-ftdi
make
sudo make install
```

Finally, an ARM toolchain is required. The ARM toolchain supplied in the Debian Wheezy repositories do not have all the required features we are after, so we must compile from source. The toolchain is based on the regular GNU toolchain, targeting the arm-none-eabi device range.

```
# Make a new folder
mkdir ~/Workspace/COSC460/Toolchain
cd ~/Workspace/COSC460/Toolchain

# Download useful GNU source distrubutions
wget ftp://ftp.gnu.org/gnu/gcc/gcc-4.7.2/gcc-4.7.2.tar.bz2
wget ftp://ftp.gnu.org/gnu/binutils/binutils-2.22.tar.bz2
wget ftp://ftp.gnu.org/gnu/gdb/gdb-7.4.1.tar.bz2
wget ftp://sources.redhat.com/pub/newlib/newlib-1.20.0.tar.gz

# Uncompress all files
tar -xf gcc-4.7.2.tar.bz2
tar -xf binutils-2.22.tar.bz2
tar -xf gdb-7.4.1.tar.bz2
tar -xf newlib-1.20.0.tar.gz

# Need to compile binutils first
cd binutils-2.22/
./configure --target=arm-none-eabi --with-cpu=cortex-m3 --with-no-thumb-interwork \
  --with-mode=thumb
make
sudo make install
cd ..

# Next is to partially compile GCC, note must include downloaded newlib libraries
# Make build folder, for out of tree compilation
mkdir gcc-build
cd gcc-build

# Build out of tree
../gcc-4.7.2/configure --target=arm-none-eabi --with-cpu=cortex-m3 --with-mode=thumb \
  --with-no-thumb-interwork --enable-languages="c, c++" --with-newlib \
  --with-headers=../newlib-1.20.0/newlib/libc/include
make all-gcc
sudo make install-gcc
cd ..

# Install newlib
cd newlib-1.20.0/
./configure --target=arm-none-eabi
make
sudo make install
cd ..

# Finish installing GCC
cd gcc-build/
make all
sudo make install
cd ..
```

```
# Finally build GDB, the most important part
cd gdb-7.4.1/
./configure --target=arm-none-eabi
make
sudo make install
cd ..
```

Please note that all software mentioned above must be successfully installed to get this research project to execute.

To run the extended Avatar framework over a vulnerable firmware, first plug in the external USB UART TTY device. Then plug in the Stellaris board. Next:

```
# Move to the firmware folder, eg small
cd ~/Workspace/COSC460/avatar-stellaris/small

# Start the framework and automatically generate an exploit
sudo python3 avatar_stellaris.py

# Output should appear in the console to indicate progress.
# Press enter to begin exploit generation when analysis is complete
```