Evaluating the limits of Symbolic Execution on Embedded Security

Zion Basque*, Nathan Smith†, Jayakrishna Vadayath‡ and Akshay Ajayan§
Arizona State University, Tempe, Arizona
Email: *zionbasque@asu.edu, †nksmith6@asu.edu, ‡jvadayat@asu.edu, §aajayan@asu.edu

***Abstract*—In this paper we explore the limits of symbolic execution in embedded devices using angr as an engine to solve most challenges in the ESC 2019 challenge set. We were able to solve all but a single challenge using angr and static analysis. The results of these challenges were that symbolic execution could solve around 50 percent of all the challenges without assistance.**

***Index Terms*—Firmware, Symbolic Execution, Static Analysis**

I. Introduction

Analysis of firmware with regards to security research has been receiving an increasing amount of attention due to the popularity of embedded devices in the modern world. The lack of exploit mitigations when compared to modern operating systems combined with the far reaching consequences of a compromised embedded device makes the exploitation of firmware a lucrative target for malicious attackers.

In this paper, we will describe approaches that we have used to analyze and exploit several firmware binaries that read data from an RFID card.

We will also describe the difficulties faced during each approach and provide an evaluation on the speed and effectiveness of each approach.

II. Background

In the sections which follow, we describe some background knowledge which the reader may find useful for the contents of the paper.

A. Symbolic Analysis

Symbolic analysis has been seen as the interaction between static and dynamic analysis as it is based on running/analyzing code in a symbolic framework. Symbolic analysis is often used in conjunction with a constraint solver. Constraint solving allows a user to determine the input needed to achieve a basic block in the programs memory.

B. Static Analysis

In static analysis, the methods used to understand the program is done solely without running the program. Often the code, or disassembly of the target binary is read by either the reverse engineer or a supporting tool to make claims about the programs logic. Static analysis is a very powerful method of analysis when there is no known method of running the target binary.

C. National Security Agency Ghidra Tool

Released in 2019 by the National Security Agency, the free and open-sourced tool Ghidra allows a user to disassemble, decompile, and graph the binary data of a target program. Ghidra supports a wide variety of processor instruction sets and executable formats and allows for analysis to take place on any platform which supports the Java Runtime Environment. Some other key features include the exposed Application programming interface (API)in Ghidra to allow users to create plug-in components and the ability for Ghidra to run in user-interactive and automated modes.

D. angr

Popularized by the University of California, Santa Barbara's capture the flag (CTF) team Shellphish, the angr project allows a user to have a framework for analyzing binaries with both static and dynamic analysis, also called concolic, analysis. This means the utility this program gives a user allows them to use a python framework to conduct symbolic solving and reverse engineering. The applications of angr have existed in research, CTF competitions, and for general binary exploitation and exploration.

E. GNU binutils

GNU Binutils is a set of binary tools targeted at binaries supported in the most common linux systems. These tools are often extremely useful when examining programs that have well documented structures, like ELFs. Common programs used in binutils for reverse engineering include strings, objdump, and readelf.

F. Firmware

Firmware shares many attributes with software. What defines the difference between firmware and software comes from how firmware interacts with the system's hardware while software will not in most cases. The firmware controls the operations of the hardware whether that hardware be a motherboard, hard drive controller, or even an Radio Frequency Identification reader.

III. Overview of Approaches

A good analysis of any binary code involves a combination of static and dynamic analysis. However, dynamically analyzing firmware binaries is an open research problem. Several

approaches have attempted to rectify this issue with varying degrees of effectiveness. [2]

The remainder of this section will be used to describe the approaches that we used to better understand and solve/exploit the different functions in each of the binaries that were provided.

A. Static analysis

The starting point of most reverse engineering methodologies is static analysis. This is usually because of the fact that static analysis is much faster than other approaches and can be used to analyze more portions of a binary than is usually possible with other approaches.

Our first step was to load the given ELF file into the NSA's Ghidra tool. Ghidra helped us gain an overview of the different functions present in the ELF. We were able to understand that the **startChallenge** function was responsible for starting the challenges that the user had chosen on the board. From this we understood that the functions which started with **challenge_** contained most of the logic for the challenges.

B. Dynamic Analysis

A common second step in reverse engineering is dynamic analysis which is usually done in a controlled environment where the reverse engineer can observe, inspect and/or modify the state of execution in order to gain better understanding of the binary. However, since the binaries that were provided used functions that were unique to the board that the firmware was running on, it was very difficult to recreate the environment so that the binaries could be executed under supervision.

Therefore, we resorted to a blind dynamic approach in which we tried different challenges against the initial sender.py file. This was done to gain more insight about how the user could select challenges as well as how to provide inputs to the board. We were able to solve one challenge using this approach.

We also tried to emulate the binary using QEMU and using the popular debugger GDB. However we were not able to successfully execute the binary in this setup.

C. Symbolic execution

When compared to the previously mentioned approaches, symbolic execution is a newer approach in reverse engineering that has been gaining in popularity. One of the most popular and open source binary analysis platforms is angr which was developed by the SecLab at University of California, Santa Barbara. angr loads binaries of different architectures into an intermediate representation upon which it performs all other operations. This allows angr to perform its analysis on all architectures supported by the underlying intermediate representation.

We found angr's symbolic execution engine to be very helpful in solving many of the challenges and will be describing our approach in the next section.

IV. Automated Analysis of Binaries

After statically analyzing a few challenge sets it became clear that challenges all take input over the same packet that is eventually put on the stack of the program. Using this idea, we were able to create a generic framework to execute any section in the provided binaries.

Listing 1. Packet structure

```
struct packet {
    char comm;
    char RFID[1024];
    char keys[48];
    char buttons;
    char challengeNum;
};
```

Listing 2. Symbolic packet

```
comm = claripy.BVS('comm', 8*1)
RFID = claripy.BVS('RFID', 8*1024)
keys = claripy.BVS('keys', 8*48)
buttons = claripy.BVS('buttons', 8*1)
challengeNum = claripy.BVS('challengeNum', 8*1)
packet = comm.concat(RFID, keys,
                     buttons, challengeNum)
```

A. Rigorously Describing Challenges

Each challenge, like any function in a program, can be described by a set of basic blocks B, where every basic block is a series of instructions with an entry and exit (jump). We can then describe three basic blocks as milestones in the succession of states: first, the basic block at the beginning of the entire binary, b_{init}; second, the basic block at the start of the procedure, b_{start}; third, the targeted basic block which successfully halts the procedure, b_{finish} such that b_{init}, b_{start}, $b_{\text{finish}} \in B$.

To keep things simple, we can say a state is a stack and a basic block: *State(B, S)*, allowing the same basic block to have multiple states. To get to the targeted basic block b_{finish}, we need to transition from $State(b_{init}, S_{init})$ to $State(b_{finish}, S_{finish})$. This transition can be seen as a set of states $T = State_0, State_1, ..., State_n$. We can reverse this sequence: $T_{\text{rev}} = State_{\text{n}}, State_{\text{n-1}}, ..., State_0$, which is a sequence to go backwards in a binary. Thus the transition from $State_{\text{finish}}$ to $State_{\text{init}}$ can be seen as solving the equation $S_{\text{n-1}} = S_{\text{n}}T$, where we attempt to transition stacks using symbolic variables to solve the equations $\forall\ State(B,S) \in Transition(State_{init}, State_{final})$. Using this idea, stacks can be broken down into small components of symbolic bits that must be transitioned (to a 1 or 0), to achieve state transitions. We setup the initial stack, S_{init}, by symbolically observing the things passed to S_{init} from S_{start}.

B. Creating Symbolic Packets

One of the biggest problems that is commonly encountered when dynamically analyzing firmware binaries is that of functions that are unique to the firmware. Since most dynamic analysis techniques start from the entry point of the binary, it takes great effort to make sure that the execution flow does not step into a function that cannot be emulated.

In this regard, one of the most useful functionalities of angr would be the ability to create a symbolic state at any arbitrary address in the binary. This is usually very inaccurate due to the loss of context sensitivity when starting analysis at arbitrary locations. However, our static analysis helped us understand that the functions whose names started with **challenge_** were the starting point of the logic of that challenge. Therefore, none of the functions that are not invoked from these **challenge_** functions are of importance with respect to that particular challenge

Our static analysis also helped us understand the format of the packet that was used as an argument to these functions. We have listed the structure of the packet in Listing 1 which we recovered with the help of Ghidra. This knowledge helped us better constrain the symbolic state which greatly reduced the possibility of path explosion during symbolic exploration.

We have described an example of how we created a symbolic state that contains a symbolic packet as its argument in Listing 2.

C. Flow Predicate Detection

During our static analysis phase, we found out that most of the functions were copying a string that was of the pattern "solved challenge xxxx abcdef" into a global character array. We understood that the objective of each challenge was to find an input that would trigger this copy functionality.

At this point, we decided to use angr's symbolic exploration which would symbolically execute each state until a state that triggered the copy functionality was reached. During the symbolic exploration, angr understands the constraints on the data and adds these constraints to the relevant bits. Once a state has reached the target address, we used angr's constraint solver to generate a concrete input that was used to reach that address.

D. Brute force method

A very common problem encountered during symbolic exploration is that of path explosion. Naive symbolic execution can use up all of the system resources even in powerful systems in cases of loops that depend upon symbolic data.

Some commonly encountered situations such as hashing are too complicated to reason about in terms of symbolic constraints. However, we found that some of the complicated situations were only using 4 or less bytes of the RFID input in the calculations. By iteratively constraining these bytes in the symbolic packet to single values, we were able to effectively fuzz the functions for possible solutions as well as evaluate the result of certain inputs without much manual analysis.

```
import angr
import claripy
import logging
logging.getLogger('angr.sim_manager').setLevel(logging.DEBUG)

# prepare packet
comm = claripy.BVS('comm', 8*1)
RFID = claripy.BVS('RFID', 8*1024)
keys = claripy.BVS('keys', 8*48)
buttons = claripy.BVS('buttons', 8*1)
challengeNum = claripy.BVS('challengeNum', 8*1)

packet = comm.concat(RFID, keys, buttons, challengeNum)
start_addr = 0x10F8+1

# prepare starting state
proj = angr.Project('./TeensyChallengeSetC.ino.elf')
state = proj.factory.blank_state(addr=start_addr)
state.regs.r0 = packet.get_bytes(0x0, 4)
state.regs.r1 = packet.get_bytes(0x4, 4)
state.regs.r2 = packet.get_bytes(0x8, 4)
state.regs.r3 = packet.get_bytes(0xc, 4)
state.memory.store(state.regs.sp, packet.get_bytes(0x10, len(packet)//8-0x10))
print(state.regs.sp)
# execute
simgr = proj.factory.simgr(state)
simgr.explore(find=0x11B8+1)
print(simgr)
import IPython;IPython.embed()
```

Fig. 1. angr framework script

One such example was the **game** challenge from Set C where we constrained 3 bytes of the RFID packet and evaluated the actions performed by the function based upon the values of those three bytes. We were able to identify a possible solution by testing different values for each of these three bytes one at a time.

V. Results

Of the provided 6 sets that contained a total of 18 challenges, we have solved all except for one challenge. The method used to solve each challenge along with the hash of the inputs are provided in Table 1.

Of the different approaches that we tried, symbolic analysis had the best ratio of effort to productivity. However, we encountered challenges which required intuition and the reasoning capabilities of a human to solve.

We tried different approaches to automatically run the provided binaries in a purely dynamic approach, but were unsuccessful in those attempts. However, we were able to successfully emulate the important functions in the binaries using a constrained symbolic execution which allowed us to identify values generated by the functions under certain conditions without a great deal of manual analysis.

On average, it took around 2-3 minutes for a purely symbolic approach to generate an input that would trigger the copy functionality in the challenges. This was preceded by 5-8 minutes of manual static analysis to estimate the complexity and gauge the effectiveness of symbolic execution in solving that particular challenge.

However, in situations where the challenge was too complex to be solved by automated methods, the time taken for manual reverse engineering was between 2-6 hours depending upon the difficulty of the challenge. Following this manual analysis, we used a constrained symbolic execution to emulate the function in order to verify the correctness of the manual solution. Even in situations where 64 bytes of RFID input was

TABLE I
CHALLENGES

Challenge set	Challenge Name	Approach	Hash
A	Stairs	angr	396f4b1cdf1cc2e7680f2a8716a18c887cd489e12232e75b6810e9d5e91426c7
	Closet	angr	8425ad5e0454e8f2398aa8a2b4a361e5670339dad91b5d81aef88fd940d7bac9
	Cafe	angr	d05235d380e913b5625d653c555de8925f249838896651a95bb35ea4e7863a5e
	Lounge	angr	6cd6ab9911818564e4f58cc5c25472a1c177917879210b077a728d96c23ccd83
B	Mobile	Static reversing + angr	f2f3792453040e837e7e1584e72859bfaa6b8c09d73d185be53b35886b6455c2
	Dance	SHA-256 decrypted online	e631b32e3e493c51e5c2b22d1486d401c76ac83e3910566924bcc51b2157c837
	Code	Bruteforce 1 byte input	372ded6746e45ef7c8ad5a22c5738a4b5aa982da66bc8a426aa1cca830d05af3
	Blue	No solution	No hash
C	Break	angr	ae4be3d07679f53cf7fd0ed9669d06bc3b22c5554c81e3bad04986bb7ab91db1
	Recess	angr	370815b8d8fde829f5c35f893d0b4139d61a775baa4181fcac1fffe014bde9ea
	Game	Static reversing + angr	63c0b41f89bbf493ba791c092b3e5473e243b9c16666f1e5eaa82bc52eeb1613
	Uno	Static reversing	b6de2d8f350d19ba182aaf3889a85fbadbc32b540547801a15fc7f88d3e85f09
D	Bounce	angr	08b243633236cba8438051b94ccd3c8c2114e1bad669db1a6d57fc069732600f
E	Steel	Hash bruteforce	5921d2ca353338c5f04c92205dc8f8bc8734f092a9e63e5f02ec106f7a7d99b4
	Caesar	Math reduction + bruteforce	551b5cff372d310b57d39b616400461be0a1450c519a2a542f33a7af0dd565f3
	Spiral	Static reversing + angr	26ee8470c732dfc821bbe0561b446dc8086560e4e222b22e6a74e559d90a7d61
	Tower	Static reversing	b019c48299dd33ec6fdc94da9d5ad06018549ee58f4a829a44d15e6980c22cbb
F	Spire	Static reversing	dd65615c86573f6416095c061e9c626c56f1720a0d2220e0d4981f508cec75d6

used in a loop whose exit condition depended on this input, this emulation took less than 2 minutes to provide a result.

VI. CASE STUDIES

In this section, we will describe some situations that we encountered during our analyses which highlight the strengths and weaknesses of different approaches.

A. Set A Lounge

Lounge was a challenge that involved an intense amount of bitshifts, variable overwriting, and moving that caused the key to this challenge to be quite obfuscated. If you were attempting to reverse this logic, it could take quite a long time.

When viewing this challenge it was clear this challenge only had one exit point, and thus made for an excellent symbolic execution solution. We simply had to load the symbolic packet using our earlier method, and direct angr to explore to the exit address 0xc22. The script took 5 minutes to write, bringing the total time spent on this challenge to 15 minutes. Indeed, this challenge had the potential to take much longer.

B. Set B Dance

This challenge uses 8 bytes of RFID as input to calculate SHA-256 hash. Then the resulting hash is compared with another hash. Win message is printed only if both the hashes are the same. We were able to find the correct input by decrypting the hash using one of the SHA-256 rainbow tables available online. Challenges involving complicated cryptographic algorithms are difficult to model in terms of symbolic constraints, and solving these will take a lot of time. Hence, tools like angr cannot be used in this scenario.

C. Set C Game

This challenge was effectively a situation where the input was used to play the popular game of tic-tac-toe. The challenge used 3 bytes from the RFID to identify the locations where the user marks the character 'o' and then uses a function to decide where the best position is to mark the character 'x' according to the users choice.

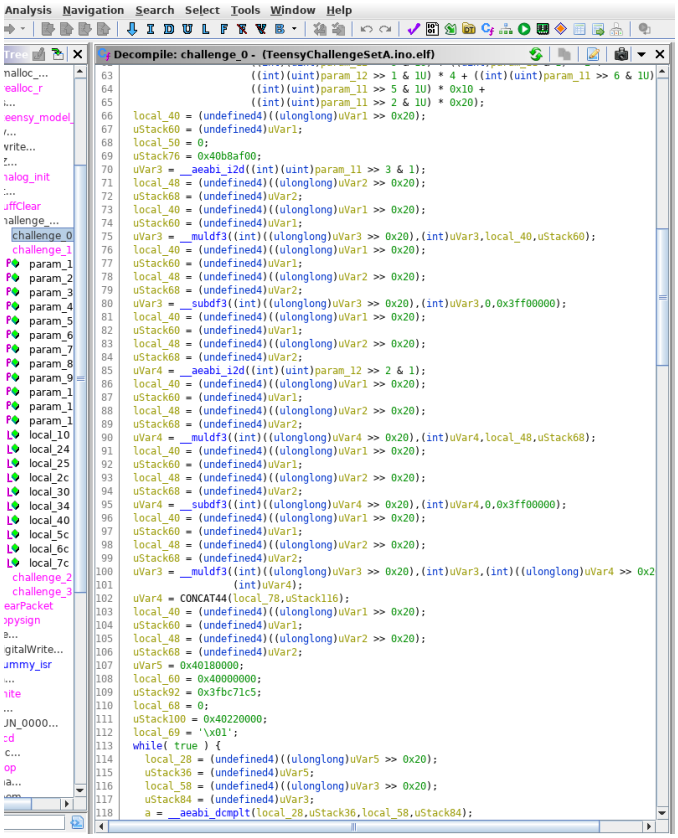

Fig. 2. Ghidra's decompilation of Lounge

The function used to make this decision is quite complex and is very difficult to manually analyze. Ideally, a dynamic analysis would have been useful in identifying the possible values returned by this function for different values of user input. However, we successfully used constrained symbolic

execution to emulate this function and identify its return value for the different inputs. Each iteration in this function used less than one minute on an average system with moderate amount of memory. This shows that constrained symbolic emulation can be used as an alternative to dynamic analysis in some cases where the context sensitivity is not a major issue.

D. Set D Bounce

This challenge was an exploitation challenge which required the user to provide inputs that would correctly overwrite a variable by using a buffer overflow. If this variable was not correctly overwritten, the function would trigger an infinite loop. The constraint on this variable was created by xoring the buttons value from the packet with one byte of the return address of the function. We were able to successfully use angr to identify a possible value for the buttons value in the packet and eventually solved the challenge by setting the value of buttons to that value on the board.

E. Set E Caeser

Caeser was an interesting challenge because it had a single exit point that would get the user the "flag", but was not solvable by angr. The reason behind this, was due to the calls back and forth between checks in the function. angr would check tests linearly in reverse order, which caused states to never complete.

To solve this challenge we used a little human ingenuity and mathematics. The base of this problem was solving the poly functions in the right order, using the following solution set. There were 4 poly functions to solve for, and 5 bytes as input – which would be a non-ideal scenario to bruteforce. The trick of this challenge was being able to solve each poly in a linear format, which meant after solving the first poly, the second poly could only have solutions in the first polys solution set and so forth. Instead of brute forcing 256^5 combinations, we were able to bruteforce around 256^2 combinations through only attempting solutions in the solution set of the poly beforehand. This is not something angr, or any current symbolic execution engine, can garner from mathematical checks in a binary.

VII. Conclusion

In this paper, we have presented the approached that we used to solve the challenges in the CSAW ESC 2019 competition. We also evaluate the strength and weaknesses of each approach that we used and a measure of the time taken for the different approaches. In our evaluation, we have found that symbolic execution engines such as the one used by angr is very useful in emulating firmware binaries and is a viable option for dynamic analysis.

However, symbolically emulating individual functions is only viable in situations where context sensitivity is not an issue. In other situations, symbolic emulation can lead to incorrect results or exhaustion of system resources.

VIII. Future work

One of the biggest difficulties that we faced during this project was the inability to inspect memory instruction by instruction in a debugger. Despite our best efforts, we were not able to emulate the binaries in QEMU with GDB attached. We will continue working towards this direction and hope to create a platform that allows users to execute arbitrary functions in a debugging environment.

Another approach will be using the popular emulation framework Unicorn alongwith symbolic execution framework angr to rehost firmware for dynamic analysis with support for debugging and instrumentation.

Improving the decompilation support for AVR and ARM architecture is another possible direction. Having better tools will make finding and fixing bugs easier.

References

[1] Yan Shoshitaishvili et al."SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis" IEEE Symposium on Security and Privacy, 2016.

[2] Daming D Chen, Manuel Edge, Maverick Woo and David Brumley "Towards Automated Dynamic Analysis for Linux-based Embedded Firmware" Network and Distributed System Security Symposium (NDSS), 2016.

[3] Drew Davidson, Benjamin Moench, Somesh Jha and Thomas Ristenpart "FIE on firmware: finding vulnerabilities in embedded systems using symbolic execution" USENIX conference on Security, 2013.