Towards Firmware Analysis of Industrial Internet of Things (IIoT)
Applying Symbolic Analysis to IIoT Firmware Vetting

Geancarlo Palavicini Jr, Josiah Bryan, Eaven Sheets, Megan Kline and John San Miguel
US Department of Defense, SPAWAR Systems Center Pacific, San Diego, California, U.S.A.

Keywords: Industrial Internet of Things, Firmware Vetting, Internet of Things, Cybersecurity, Vulnerability Research, Embedded Systems, Security, Malware, Emerging Threats, Binary Analysis, Virtualization.

Abstract: Embedded systems and Industrial Internet of Things (IIoT) devices are rapidly increasing in number and complexity. The subset IIoT refers to Internet of Things (IoT) devices that are used in manufacturing and industrial control systems actively being connected to larger networks and the public internet. As a result, cyber-physical attacks are becoming an increasingly common tactic employed to cause economic and physical damage. This work aims to perform near automated firmware analysis on embedded systems, Industrial Control Systems (focusing on Programmable Logic Controllers), Industrial Internet of Things devices, and other cyber-physical systems in search of malicious functionality. This paper explores the use of binary analysis tools such as angr, the cyber reasoning system (CRS) 'Mechanical Phish', American Fuzzy Lop (AFL), as well as virtualization tools such as OpenPLC, firmadyne, and QEMU to uncover hidden vulnerabilities, find ways to mitigate those vulnerabilities, and enhance the security posture of the Industrial Internet of Things.

1 INTRODUCTION

Embedded systems and Industrial Internet of Things (IIoT) devices are rapidly increasing in number and complexity. Industrial Internet of Things devices are a part of a large family of Internet of Things (IoT) devices, which are simply non-traditional devices that are now being connected to the public Internet. The subset IIoT refers to IoT devices that are used in manufacturing and industrial control systems. As a result, cyber-physical attacks are becoming an increasingly common tactic employed to cause economic and physical damage (Sadeghi et al., 2015). Paired with the lack of efficient cyber security analysis, increased connectivity, sloppy programming, and speed to market pressures, cyber-physical attacks have created a dangerous climate for Operation Technology (OT) networks and Internet of Things devices.

As embedded technology and capabilities increase, the firmware for these systems becomes increasingly difficult to analyze in search for malicious functionality. Our goal is to perform near automated firmware analysis on embedded systems, Industrial Control Systems (focusing on Programmable Logic Controllers), Industrial Internet of Things devices, and other cyber-physical systems in search of malicious functionality. Our work adapts UC Santa Barbara's binary analysis framework called 'angr' (Shoshitaishvili et al., 2016), as well as components from their cyber reasoning system (CRS) called 'Mecanical Phish' to perform semi-automated analysis on IIoT systems. This paper makes the following contributions:

- Leverages proven open-source technologies and approaches for tradtional software / firmware analysis for use on IIoT firmware
- Extends the work of Shoshitaishvili et al. to incorporate custom architecture backends for non-standard and propritary architectures to the angr framework
- Lays out a proposed approach for automating portions of our firmware analysis approach

This paper is organized as follows. Section 2 discusses the current state of the art for IoT binary analysis. Section 3 dives into the approach taken to analyze IIoT devices in search of vulnerabilities and the effort to automate the processes, as well as challeges and mitigations. Section 4 discusses the initial results and findings of applying dynamic symbolic execution and symbolic assisted fuzzing to analyzing IIoT device firmware. Section 5 summarizes the paper with the conclusion and Section 6 closes out the work presented in this paper with a glance toward future work.

Jr, G., Bryan, J., Sheets, E., Kline, M. and Miguel, J.
Towards Firmware Analysis of Industrial Internet of Things (IIoT) - Applying Symbolic Analysis to IIoT Firmware Vetting.
DOI: 10.5220/0006393704700477
In *Proceedings of the 2nd International Conference on Internet of Things, Big Data and Security (IoTBDS 2017)*, pages 470-477
ISBN: 978-989-758-245-5

2 RELATED WORK

The runtime-monitoring framework presented by (Janicke et al., 2015) addresses subtle changes to critical system behavior through semantic attacks. Traditional protection systems such as Intrusion Detection Systems (IDS) have difficulty identifying such classifications of attacks. These subtle changes create states in program execution that could result in machinery violating safety requirements such as a drill being used on a product that does not exist at the location that drilling occurs. The state execution of the machinery specifies the current state of the system, and the behavior that should be exhibited from the next state. Behavior outside of this specification is determined to be outside of the normal execution, and thus malicious.

Cruz et al., present Shadow Security Unit (SSU) in (Cruz et al., 2015) as an in-line network monitor. The lightweight device sits in parallel to a PLC or RTU and passively collects communications and control data to detect active attacks in a live environment. This is presented as a solution to the specific challenge of sensitivity to timing that is present with ICS and SCADA systems and it is a contibution to the larger work the CockpitCI project as described in (Cruz et al., 2014). SSU is a minimally invasive behavior monitor and correlation engine for anomaly detection and foucses on live run-time environments.

The work most closely related to ours is that of Almgren et al., under the CRISALIS Consortium (Almgren et al., 2015). They are working on automated vulnerability discovery for Critical Infrastructure (CI) environments. Their approach includes application-level protocol fuzzing, emulation and dynamic analysis of embedded devices, and security testing prioritization by means of vulnerability indicators. They outlines 5 challenges to large scale vulnerability analysis of embedded firmware, namely 'building a representative dataset', 'firmware identification', 'unpacking and custom formats', 'scalability and computational limits', and 'results confirmation' (Costin et al., 2014). Our work aims to ease two of those challenges, namely the 'unpacking and custom formats', as well as 'results confirmation'. We're tackling the 'unpacking and custom formats' challenge through the development of custom architecture backends for the angr framework, and as much as possible, automating our extraction process. We are also easing the challenge of "results confirmation' through our emulation of firmware to verify the exploitability of discovered vulnerabilities. A major difference in our approach stems from our leveraging of static, dynamic, and symbolic analysis of the firmware samples with the angr framework, as well as symbolic-assisted fuzzing through Driller's AFL/angr hybrid approach.

Mclaughlin et al., introduce the Trusted Safety Verifier (TSV) in (McLaughlin et al., 2014) which supports control system security by reducing the trusted computing base for safe process execution throughout the entire system. TSV is deployed as an in-line device that uses an instruction list lifter and translates it to Instruction List Intermediate Language (ILIL) for ease of processing. It is designed to handle PLC specific features, like function blocks, timers, counters, master control relays, data blocks, and edge detection in order to detect potential for data injection attacks and PLC firmware exploitation.

3 APPROACH

Our proposed approach for automated firmware analysis focuses on three major tasks preparation of the firmware image for loading into the angr framework, emulation for verification of discovered vulnerabilities, and analysis of the firmware sample 'angr style'. Once loaded in the framework, we can benefit from the angr's reliance on the python programming language to further automated the process. The stages of our approach are as follows (Figure 1 shows a graphical representation of this process):

- Extraction and cleanup;
- Emulation;
- Analysis angr style.

The firmware must first be extracted and loaded into the angr framework. For the extraction portion we will make use of tools such as binwalk (devttys0, 2016a) and firmware-mod-kit (Collake and Heffner, 2013) to extract the packaged firmware.

Once the firmware is extracted we will run our own developed software to add or remove content from the binary to prepare it for efficient analysis. Cleaning up the packaged firmware is extremely important for utilizing many of the resource intensive analyses in angr. An example of this is the Symbolic Execution portion, which is prone to path explosion while analyzing complex binaries, which we will further discuss in section 3.5.

Once the firmware has been successfully extracted and cleaned up, we will emulate the firmware using OpenPLC (for PLC emulation) (Alves et al., 2014), QEMU (Bellard, 2017), and/or Firmadyne (Chen et al., 2016). After these initial steps, it will be time to load the firmware into angr using the framework's default loader CLE or utilizing IDA's binary loader. Once the binary is loaded we can use a combination

of various binary analysis techniques (E.G. fuzzing and symbolic execution) to discover the functionality and identify malicious behavior such as backdoors, information leakage, or botnet code.

To aid the analysis, we will be adding vendor specific conventions and libraries to the knowledge base that angr populates after recovering a Control Flow Graph (CFG) of the firmware. Angr's knowledge base is a shared repository of information discovered by the angr framework as it progresses through the analysis of a particular sample(Shoshitaishvili et al., 2016).

3.1 Firmware Extraction

Extraction of firmware from IIoT devices makes it possible to replicate the device's behavior through emulation.

Three techniques are explored, which include downloading the firmware from the vendor's website or additional sources, capturing it during a device update, and extracting from the device.

Accessing firmware images through vendors can be a trying process, due to protection mechanisms implemented by manufacturers. Given the vast number of devices to analyze, this method is not feasible, as it suffers a lack of scalability. The remaining feasible options are constrained by physical possession of the IIoT device.

IIoT firmware updates are almost exclusively applied automatically, which can lead to hurdles in firmware retrieval based on how the updates are pushed to the device. The challenge of extracting a working firmware image from the device is not an uniform process and in many instances unique to the specific device.

Potential interfaces to extract the firmware consist of JTAG, UART, and in-circuit serial programming (ICSP). Vendors take steps to block debug interfaces such as the ones listed above; however, dumping the flash chip directly may be the only option.

With the firmware image extracted, the task turns to extracting the core components: boot loader, kernel image, and file system. There is a wide spectrum of vendors with no explicit standard on how firmware images should be structured. The consequence is having to reverse engineer and analyze each component to determine what is pertinent to the system for emulation.

Luckily, Linux provides utilities and tools to aid in this endeavor. The Linux file utility will verify the contents of the image to either be a compressed file or data. Running further Linux utilities `strings` and `hexdump` can reveal insightful information such as firmware version, operating system, and boot loader. The information provided by these utilities contributes to a preliminary blueprint of the operation of an IIoT device.

Binwalk, a firmware analysis tool (devttys0, 2016a), can be executed against the firmware image to identify embedded files, executable code, and perform recursive file system extraction. Two Linux file systems commonly associated with IIoT device firmware are squashfs (devttys0, 2016b), a compressed read-only file system, and jffs2 (Gupta, 2016) (OWASP, 2016), a long-structured file system for use with flash memory devices.

There are two open-source projects, sasquatch (devttys0, 2016c) and jefferson (sviehb, 2016) respectively, that add modifications to existing decompression utilities for the file systems. The file system is composed of binaries and initialization scripts that can be used to investigate disassembly for static analysis and emulate behaviors of the IoT device for dynamic analysis.

In addition, vendors modify file systems to prevent them from being extracted. 'Firmware modification kit' is designed to attempt the extraction of untraditional squashfs and cramfs file systems that have been modified from firmware using TRX or uImage headers. This tool is critical to the extraction process due to its ability to rebuild what has been disassembled. Alterations can be made to exhibit malicious intent for the exploitation of the rebuilt device firmware. The tool has the capability to re-flash the IIoT device with the malicious rebuilt binaries, and can be verified through emulation.

3.2 Firmware Emulation

Firmware emulation provides an operational environment separated from device hardware. QEMU is a machine emulator and virtualization platform, that includes capabilities for full system and user-mode emulation (Bellard, 2017). This is achieved by QEMU through hardware virtualization capable of emulating CPUs with dynamic binary translation. Various CPU architectures are supported for emulation including IA-32, x86-64, MIPS, and ARM. QEMU is the standard emulation tool for firmware binaries and images, and can be leveraged for deep analysis of firmware behavior. Understanding the disparity in behaviors of non-malicious and malicious binaries is essential to determining potential vulnerabilities in firmware. With increased popularity of IIoT devices, assurance in non-malicious behavior is vital in securing these devices from adversaries requiring emulation of the underlying firmware for verification of any discovered vulnerability.

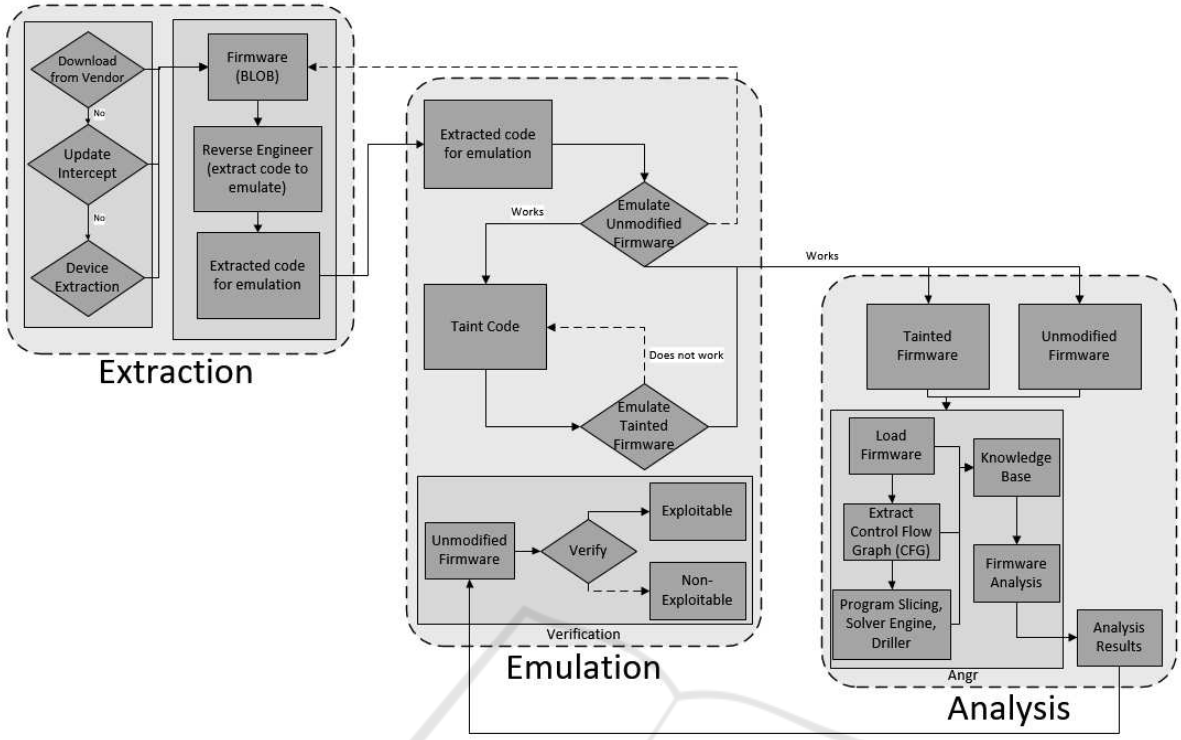

Figure 1: Approach: Extraction & cleanup, Emulation, Analysis angr style.

3.2.1 Emulating PLCs

Due to the nature of PLC deployments as 'off-network' devices, security was not an area of focus for Programmable Logic Controllers (PLCs), the control unit for ICS. This phenomenon of merging physical devices such as PLCs with the internet has been coined the Industrial Internet of Things. The concept of IoT complements the functionality of a PLC by controlling other machines including sensors and other devices across a network with the desire to be autonomous. The main test subjects for this research are PLCs, both commercially available PLC devices, as well as open-source technologies and PLC virtualization platforms.

OpenPLC is a fully functional, standardized, open-source PLC (Alves et al., 2014), capable of supporting all 5 of the programming laguages in the IEC-61131-3 specification (ST, IL, LADDER, FBD and SFC). It allows research to be done on physical hardware and processes in conjunction with an environment for emulation such as Linux. The project includes a Modbus/TCP communication capability that can interface with any human machine interface (HMI) software that supports Modbus/TCP, it includes a nodeJS environment for interfacing OpenPLC with the hardware. Modbus is a serial communication protocol designed by Modicon, and commonly used for PLC communication (Modbus, 2012).

3.3 angr Framework

The angr framework will be used for its unique ability to combine static, dynamic, and symbolic analysis. This research leverages angr's capabilities applied towards Programmable Logic Controller (PLC) firmware, with the goal of discovering any hidden vulnerability in the underlying software controlling the hardware, known as the firmware. Although angr's loader (CLE) can handle most common hardware architectures, PLC firmware images can pose interesting challenges with respect to this task. Therefore, before we can leverage the binary analysis capabilities of the angr framework, we must first overcome the challenge of loading the PLC firmware image into angr for further processing.

The angr framework is a python based tool for analyzing binaries developed by researchers from UC Santa Barbara. It is a product of the Defense Advanced Research Project Agency's (DARPA) Vetting Commodity IT Software and Firmware (VET) program and further enhanced through DARPA's call for autonomous cybersecurity systems known as the Cyber Grand Challenge (CGC) (DARPA, 2016). The angr development team finished in the top three for the CGC final competition. The goal of the CGC

was to achieve autonomous systems capable of testing for vulnerabilities, exploiting the vulnerabilities found, generating security patches, and applying those patches.

UCSB's cyber reasoning system (CRS), Mechanical Phish (Shellphish, 2016), implemented angr with American Fuzzy Lop (AFL) (lcamtuf, 2017) to achieve the challenges set forth by DARPA. The ability of angr to employ automation of a collection of analyses makes it an ideal tool for analysing firmware behavior.

Angr is comprised of four main components: CLE, VEX, Claripy, and Simuvex. The framework incorporates a binary loader, CLE, responsible for making the binary easy for angr to analyze, whether static, dynamic, or symbolic analysis is to be performed.

Loaded binaries are converted to an intermediate representation (IR), Valgrind's IR 'VEX'. This conversion abstracts away differences in architecture to allow a single analysis to be run on all loaded binaries.

The solver engine, Claripy, is used for constraint solving of concrete and symbolic expressions which are necessary for symbolic execution.

The final component, the simulation engine Simuvex, provides a semantic understanding of VEX IR in combination with program state in order to accomplish static, dynamic, and symbolic analysis.

These four components are fundamental to allowing angr to perform dynamic symbolic execution and various static analysis on binaries with an easy to use extensible framework.

3.3.1 Firmware Loading with angr

Correctly loading the firmware into angr is the first and arguably most important part in our process of analyzing embedded systems. By default angr's loader CLE can load most common architectures including ARM, MIPS, x86, AMD, AARCH, and PPC (Shoshitaishvili et al., 2016).

If the binary we are analyzing is not one of these architectures (for example a binary blob), we can extend CLE to perform more fine grained entry point discovery where we can begin analysis. This will require us to both develop and integrate our own software and other open source tools to aid CLE.

One technique that angr currently supports is making use of IDA's loader to load binaries that CLE cannot, and leveraging the feedback from IDA to begin analysis. Any vendor specific information that we find when loading binaries will be added to the knowledge base to aid future loading and entry point discovery efforts.

3.3.2 Firmware Analysis angr Style

Angr offers many different analysis techniques that can be performed on a loaded binary. For our purposes we will focus on Control Flow Graph (CFG) recovery, Program Slicing, the solver engine, AFL, and Symbolic Execution (Shoshitaishvili et al., 2015).

Without a CFG most of the other analysis will not work, and recovering a CFG from complex binaries can be quite difficult. This further stresses the importance of narrowing down our areas of interest within a binary, creating a comprehensive firmware knowledge base, summarizing as much code as possible (such as Standard C Libraries), and removing unnecessary content.

Once we have extracted the CFG we will perform fuzzing and symbolic execution. For this portion we will leverage Driller, which augments fuzzing with symbolic execution to discover paths within the program which lead to an authenticated state, while recording the constraints required to reach that state. Driller switches back and forth between fuzzing and symbolic execution to work its way through a program (Stephens et al., 2016). The fuzzing portion is handled by AFL, while the symbolic execution is handled by angr.

For example, if a backdoor existed within the firmware of a smart plug, driller could be used to find the backdoor along with the constraints required to reach that authenticated state, and use the built-in solver engine to find the input required to access the backdoor based on those constraints.

Another feature of angr that we plan to make use of is program slicing. Program slicing is a subset of statements from the original program. While analyzing embedded systems, there are certain parts of the system that we are more concerned with than others, which is where program slicing comes in. This gives us the ability to focus exclusively on a particular piece of the program.

3.3.3 Driller

In this section we give a brief description of the hybrid symbolic-assisted fuzzing approach implemented by UC Santa Barbara in Driller (Stephens et al., 2016), it must be noted that we do not take any credit for their implementation.

Driller uses an instrumented fuzzing engine to drive the dynamic symbolic execution, once the fuzzer reaches a point where it cannot find any other paths, it switches to angr's symbolic execution engine to leverage its ability to find the values needed to satisfy the constraints of the branches that the fuzzer cannot solve. It then switches back to the AFL fuzzer

with the new inputs need to get through the next portion of the binary. This process takes place as many time as needed to reach a program crash point.

One of the strengths of driller's fuzz-guided symbolic execution is that it does not require test cases to be supplied at the start of the analysis, although it speeds up the analysis if you supply AFL with initial test cases. It can generate its own input test cases leveraging angr's symbolic execution engine.

The concept of combining fuzzing with symbolic execution is not novel, but Driller's ability to automatically test portions of the code to replace them with the symbolic summaries without user intervention is a novel approach that we are leveraging to aid in automating the analysis of IIoT firmware. Interestingly, this approach deals with both the path explosion challenge as well as the challenge of automating any and all possible portions of our approach, which we'll cover in the next subsection.

3.4 Challenges and Mitigations

There are two overarching challenges faced by our current approach. The first challenge is inherent in any solution based on dynamic symbolic execution, namely the path explosion problem. The second challenge stems from the lack of automation in both extraction and analysis of firmware images. We'll cover both of these challenges and the chosen mitigations in the following subsections.

3.4.1 Path Explosion

The path explosion problem is a well-known limitation of concolic execution or dynamic symbolic execution approaches(Shoshitaishvili et al., 2016). Anytime the analysis engine reaches a branch, it solves the constraints required to take both sides of the branch. As the analysis engine discovers an ever-growing number of branches (and solves the constraints required to take both paths of each branch), the number of paths begins to grow at an exponential rate (Shoshitaishvili et al., 2016). Several approaches have been proposed in the literature to mitigate the path explosion problem, such as program slicing (**?**), instrumenting the symbolic analysis engine (Stephens et al., 2016), path merging, under-constraint symbolic execution (Shoshitaishvili et al., 2016).

We are investigating the efficacy of program slicing and fuzz-guided symbolic analysis provided by the angr framework and the driller hybrid approach developed by UCSB (Stephens et al., 2016), as well as reducing complexity and the amount of code analyzed with symbolic summaries.

Symbolic summaries are not a new concept in binary analysis (Stephens et al., 2016). They are manual descriptions of the state changes cause by a given function to a particular program execution. They summarize the expected output and end result of executing the function, expressed to the analysis engine in similar form to a binary instruction, thus allowing the analysis to skip that particular function's code. Instead of having to analyze the entire function to reach that changed state, the summary is supplied to the analysis engine.

Symbolic summaries help the analysis by simplifying the amount of code to be analyzed, reducing the complexity of the analysis. It enables the analysis to drive deeper into the program and aids in mitigating some of the path explosion issues inherent in symbolic execution based approaches.

The one drawback from symbolic summaries is that since those portions of code are not analyzed, it's possible that a flaw in that summarized portion of the code will be missed. This is a trade-off that we accept in order to reach into deeper portions of the firmware. The other technique relied upon by our work is the use of automated fuzz-guided symbolic analysis approach, or symbolic-assisted fuzzing, embodied by the Driller solution developed by UC Santa Barbara for the DARPA VET and CGC programs (DARPA, 2016) [discussed in section 3.3.3].

3.4.2 Process Automation

As we discussed in section 3, firmware extraction can be a difficult and at times an extremely manual process. Part of the challenge of automating this process is due to the fact that PLC firmware can come in many different specialized architectures, and proprietary implementations that can complicate extraction, emulation and analysis e.g., Asymmetric Multi Processing architecture, specialized instruction set architecture, ARM-Cortex. Although the angr framework's loader (CLE) can load binaries from several different architectures, including ARM, it can have problems loading custom firmware samples. The framework has little to no support for SPARC, PIC or AVR, among other specialized embedded systems architectures.

One of our approaches to mitigating this challenge is through the development of additional architecture support for the angr framework. The use of the framework in itself allows us to develop python scripts to automate our process. In combination with the AFL fuzzer, as it is implemented in, the framework further automated the discovery of vulnerabilities, which we leverage to further our automation efforts. As such, we are in the early stages of development of custom

architecture backends for extending angr's ability to load PLC firmware easily and as automated as possible.

For this task we are focusing on the following steps: (angr, 2017)

1. Adding the architecture information to the appropriate files in the angr framework.
2. Adding an intermediate representation (IR) translation to work with VEX IR. This may be either an extension to PyVEX, producing IRSBs,
3. If your IR is not VEX, add a simuvex.SimEngine to support it.
4. Adding a calling convention (simuvex.SimCC) to support SimProcedures (including system calls)
5. Adding or modifying an angr.SimOS to support initialization activities.
6. Creating a CLE backend to load binaries, or extending the CLE ELF backend to know about the new architecture if the binary format is ELF.

The other approach involves developing modules and scripts for as much of the process in our manual extraction of firmware images. Our exploration and use of the firmadyne solution by (Chen et al., 2016) attempts to leveraging previous solutions aimed at automating this difficult task. As we continue to explore the automation of PLC firmware extraction and analysis, we will continue pursuing the potential expansion of firmadyne to further automate the tools and extend its capabilities.

4 INITIAL RESULTS

A prototype system was developed in order to perform early analysis of Programmable Logic Controller firmware vetting utilizing the angr framework. The prototype system leverages the OpenPLC (Alves et al., 2014) project on a Raspberry PI 3 Model B. The OpenPLC project can be used to emulate a similar process running on a Siemens S7-1200 PLC.

The prototype process is composed of the emulated PLC, running a ladder logic program that controls the on/off functions of a fan, as well as the speed that the fan operates under. We performed an angr-base analysis of both the extracted firmware image and the ladder logic program controlling the process.

We successfully extracted the PLC's firmware. Recalling, from our discussion on our use of emulation in section 3, that the purpose of the emulation step, in our approach, is to aid in the verification of any discovered vulnerability, and to help us determine the viability of exploiting the vulnerabilities found. Given that we already have the emulated process, we will rely on this process to verify the exploitability of any discovered vulnerabilities.

Once the firmware is extracted, we loaded the firmware sample image into the angr framework for analysis. We conducted analysis on the sample, including dynamic symbolic execution analysis. Using the angr framework, we extracted and added to the framework's knowledge base, a data dependency graph, and a Control Flow Graph recovery including:

```
- function list
- node list
- predecessors list
- successors list
```

We also identified the locations of the ladder logic program, within the firmware image, that controls the fan speed (in the OpenPLC/Raspberry Pi prototype). The analysis showed a lack of stack protection mechanisms, such as Data Execution Protection (DEP) or Address Space Layout Randomization (ASLR). This lack of protection mechanisms makes the class of stack protection vulnerabilities, such as overflows, a possible attack vector.

The second major observation from the analysis results was the existence of a authentication bypass vulnerability similar to the Siemens SIMATIC S7-1200 PLC Systems Replay Security Bypass and Denial of Service Vulnerabilities (Cert, 2014) (Beresford, 2011). In terms of this vulnerability, we can conclude that any process that writes to the modbus coil will be accepted as valid input (allowing changes to the fans operations and speed). Specially crafted packets (in our case modbus packets) would allow an attacker to send packets to the program and change values in the registers, and the process would be changed based on the false values provided by the attacker.

There are some differences between our emulated process and an S7 PLC that we would like to point out. OpenPLC uses modbus as its communication method, whereas the public exploits for the S7 operate against the iso-tsap protocol for communications. Thus, Siemens PLCs with older firmware version are vulnerable to replay attacks over iso-tsap, whereas OpenPLC is vulnerable to replay attacks over modbus.

5 CONCLUSION

This work leverages the binary analysis framework 'angr', portions of the cyber reasoning system (CRS) 'Mechanical Phish' (Shoshitaishvili et al., 2016)

(Shoshitaishvili et al., 2015) (Shellphish, 2016), as well as firmware extraction and modification techniques and tools to automate the discovery of vulnerabilities in IIoT devices. We have chosen to use PLCs as our initial IIoT test subject.

Our approach includes extraction and emulation of PLC firmware, as well as analysis using angr, AFL, and Driller. This approach has helped us uncover vulnerabilities, enabling us to devise solutions to mitigate those vulnerabilities in order to enhance the security posture of the Industrial Internet of Things. We have some early results that have been able to discover vulnerabilities in Industrial Internet of Things emulated in our laboratory environment, namely lack of stack protection and authentication bypass. As more analyses are conducted and verified, we will update the community on findings and proposed mitigations to the discovered vulnerabilities.

6 FUTURE WORK

Given the early result discussed in the paper, we have begun expanding our analysis of PLC firmware on several brands of controllers. We have started to analyze a few versions of the Siemens S7 controller, as well as several different models of Allen Bradley PLCs. We are also exploring the potential to improve the performance of angr through the use of symbolic summaries. We are working towards expanding the angr framework's ability to load other architectures specific to PLC manufacturers, and exploring the potential to extend the firmadyne tool to further automate the analysis of PLC firmware.

REFERENCES

Almgren, M., Balzarotti, D., Stijohann, J., and Zambon, E. (2015). Runtime-monitoring for industrial control systems. *Electronics*, 4(3):995 – 1017.

Alves, T. R., Buratto, M., de Souza, F. M., and Rodrigues, T. V. (2014). Openplc: An open source alternative to automation. In *Proc. IEEE Global Humanitarian Technology Conf. (GHTC 2014)*, pages 585–589.

angr (2017). angr-docs. Contributing to the framework.

Bellard, F. (2017). Qemu.

Beresford, D. (2011). Siemens simatic s7-1200 plc systems replay security bypass and denial of service vulnerabilities.

Cert, I. (2014). Siemens s7-1200 plc vulnerabilities.

Chen, D. D., Egele, M., Woo, M., and Brumley, D. (2016). Towards automated dynamic analysis for linux-based embedded firmware. In *ISOC Network and Distributed System Security Symposium (NDSS)*.

Collake, J. and Heffner, C. (2013). Firmware modification kit.

Costin, A., Zaddach, J., Francillon, A., Balzarotti, D., and Antipolis, S. (2014). A large-scale analysis of the security of embedded firmwares. In *USENIX Security*, pages 95–110.

Cruz, T., Barrigas, J., Proença, J., Graziano, A., Panzieri, S., Lev, L., and Simões, P. (2015). Improving network security monitoring for industrial control systems. In *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*, pages 878–881. IEEE.

Cruz, T., Proença, J., Simões, P., Aubigny, M., Ouedraogo, M., Graziano, A., and Yasakhetu, L. (2014). Improving cyber-security awareness on industrial control systems: The cockpitci approach. In *13th European Conference on Cyber Warfare and Security ECCWS-2014 The University of Piraeus Piraeus, Greece*, page 59.

DARPA (2016). Darpa cyber grand challenge.

devttys0 (2016a). Binwalk. Firmware Analysis Tool.

devttys0 (2016b). Reverse engineering firmware: Linksys wag120n. SquashFS common file system for IoT.

devttys0 (2016c). Sasquatch. Set of patches to the standard unsquashfs utility.

Gupta, A. (2016). Firmware analysis for iot devices.

Janicke, H., Nicholson, A., Webber, S., and Cau, A. (2015). Runtime-monitoring for industrial control systems. *Electronics*, 4(3):995 – 1017.

lcamtuf (2017). American fuzzy lop.

McLaughlin, S. E., Zonouz, S., Pohly, D., and McDaniel, P. (2014). A trusted safety verifier for process controller code. In *NDSS*, volume 14.

Modbus (2012). *MODBUS Protocol Specification*. Modicon, v1.1b3 edition.

OWASP (2016). Iot firmware analysis.

Sadeghi, A. R., Wachsmann, C., and Waidner, M. (2015). Security and privacy challenges in industrial internet of things. In *Proc. 52nd ACM/EDAC/IEEE Design Automation Conf. (DAC)*, pages 1–6.

Shellphish, U. (2016). Mechanical phish. Cyber Reasoning System for DARPA Cyber Grand Challenge.

Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C., and Vigna, G. (2015). Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Proceedings of the 2015 Network and Distributed System Security Symposium*.

Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., and Vigna, G. (2016). Sok: State of the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy*.

Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., and Vigna, G. (2016). Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the 2016 Network and Distributed System Security Symposium*.

sviehb (2016). Jefferson. JFFS2 filesystem extraction tool.