THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

COLLEGE OF INFORMATION SCIENCE AND TECHNOLOGY

A Survey of Firmware Testing and Analysis Tools Used on Embedded Devices

RAED FAROUQ M KATIB
SPRING 2022

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree
in Information Science and Technology
with honors in Information Science and Technology

Reviewed and approved* by the following:

Peng Liu
Professor of Cybersecurity
Thesis Supervisor

Marc Friedenberg
Assistant Teaching Professor
Honor Adviser

* Electronic approvals are on file.

ABSTRACT

The number of embedded devices, or IoT devices, used in the market is increasing dramatically. By 2025, the estimated number of IoT devices used worldwide will be about 30.9 billion. However, the evolvement of these devices makes organizations an increasingly high-security concern. In 2019, the estimated percentage of attacks on IoT devices that impacted critical operations was 33 percent. In addition, IoT devices, unlike traditional PCs, has minimal system requirement, making them more challenging to update and secure. The threat to embedded devices' security primary is due to their different architecture compared to traditional PCs that security practitioners started using firmware testing and analysis tools to overcome. In this paper, we plan to extend the analysis of firmware testing tools in the context of the embedded device to include newly developed tools with some old tools analyzed in the past. We also plan to examine other factors that might contribute to the development of new tools. We consider a few tools that managed to discover new vulnerabilities and compare them based on the criteria we set. Essentially, we provide our evaluation of the tools and present the key findings.

TABLE OF CONTENTS

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank Professor Peng Liu for providing me the opportunity to learn and work on embedded devices concerns. At first, I was not sure about the topic that I must choose, but he guided me to some of the interesting topics I even learned. I would also thank him for motivating and supporting me during the thesis writing process.

I would also like to thank Dr. Wei Zhou from the National Computer Network Intrusion Protection Center for helping me during the project. Professor Peng introduced me to Dr. Zhou when I first started working on the assignment. Dr. Zhou was the one who developed the firmware testing tool that I used to learn most of the things related to firmware testing. Indeed, I started the work on the thesis with zero experience in firmware testing and embedded devices security. Dr. Zhou guided me to the resources that I needed to learn everything since I started.

Thank you all.

Chapter 1

Introduction

The number of embedded devices, or IoT devices, used in the market is increasing dramatically. By 2025, the estimated number of IoT devices used worldwide will be about 30.9 billion (*Global IoT and non-IoT connections 2010-2025*, n.d.). Embedded devices helped many organizations in crossing their barriers and achieving better business values. However, the evolvement of these devices makes organizations an increasingly high-security concern. In 2019, the estimated percentage of attacks on IoT devices that impacted critical operations was 33 percent (*IoT security concerns worldwide 2019*, n.d.). In addition, IoT devices, unlike traditional PCs, has minimal system requirement, making them more challenging to update and secure. The current tools used to make the traditional PCs more robust are merely applicable to embedded devices due to their distinct behavior, leading to more threats to organizations that adopt embedded devices in their systems.

The threat to embedded devices' security primary is due to their different architecture compared to traditional PCs. Some malware and attacks aimed to exploit embedded devices, such as the Mirai botnet and the Reaper, behave in different ways compared to traditional PCs malware (Antonakakis et al., 2017; *Reaper*, 2017). Due to the high-security concerns of embedded devices, researchers conducted studies and explored these concerns. In essence, finding bugs in embedded devices became a major research topic, where researchers apply firmware testing to embedded devices. Many researchers designed firmware testing and analysis tools and frameworks in order to make embedded devices robust, including Costin et al.

(2014), Feng et al. (2020), Mera et al. (2021), Fowze et al. 2021, and Zhou et al. (2021). Most of the studies conducted in the embedded device security field focused on firmware analysis, including static analysis, dynamic analysis, fuzzing, and different methodologies for testing embedded devices' firmware. Their primary goal is to detect vulnerabilities in embedded devices to enhance their security. Also, some of the designed tools focus on the analysis of the current status of the firmware and improving its day-to-day performance while avoiding threats.

The emergence of firmware testing in embedded devices security improvement led to the development of various tools available widely during the last decade (Ai et al., 2020; Baldoni et al., 2018; Feist et al., 2016; Feng et al., 2020; Fowze et al., 2021; Gustafson et al., 2019; Mera et al., 2021; Niesler et al., 2021; Oser et al., 2020; Zaddach et al., 2014; Zhou et al., 2021). Each tool focuses on analyzing the firmware of the device, following different methods to analyze embedded devices. Some researchers also surveyed the existing embedded devices testing tools and compared them (Qasem et al., 2021; Wright et al., 2021).

However, most researchers are concerned about embedded devices' security focus on firmware testing and analysis of these devices, putting little effort into analyzing other factors that might lead to decreasing the effectiveness of the tool, such as execution paths analysis. In our research, we plan to extend the analysis of Qasem et al. (2021) and Wright et al. (2021) to include newly developed tools with some old tools that follow different approaches than the tools discussed by them. We also plan to examine other factors that might contribute to the development of new tools. For instance, some tools use different methods for analyzing the firmware of embedded devices that might be combined to enhance the performance of the firmware testing and analysis. Also, we will examine the coverage of the dataset of each tool. By

using the word dataset, we do not mean the firmware sample but the execution paths generated by running each firmware image in the firmware sample.

The purpose of this paper, surveying firmware testing tools, is to extend the analysis of the current firmware tools to cover more factors that could contribute to improving firmware testing and analysis in future tools. First, we review the embedded devices concerns that researchers addressed, and did not address, that led to the existing threats on embedded devices. Second, we review and summarize the existing embedded devices testing and analysis methods and tools designed by other researchers generally. Third, we present our approach for the analysis of the new tools that would extend on the surveys of Qasem et al. (2021) and Wright et al. (2021) approaches by adding more factors and new tools. Fourth, we conclude our findings from the surveyed firmware analysis tools and compare the new tools with some of the previously examined tools by other researchers. We then summarize our key findings from the comparison. Finally, we would bring attention to some issues that were not addressed in the past that could be addressed in future research.

Chapter 2

Embedded Devices Security Concerns

Addressing embedded devices vulnerabilities is different from traditional PCs issues. The Reaper and the Mirai botnet are some IoT-related vulnerabilities presented by a few researchers (Antonakakis et al., 2017; *Reaper*, 2017). Their research shows the different methodologies security professionals followed to address the issues. The Mirai botnet attack drew attention to the lack of security practices for embedded devices on the internet (Antonakakis et al., 2017). It showed that millions of embedded devices were comprised in a short time by the exploitation of different protocols. On the other hand, the reaper botnet presented various concerns for embedded devices. Most embedded devices are based on hard-coded programs written directly to the firmware, including crucial information such as passwords and root access credentials (*Reaper*, 2017). With the high number of embedded devices, managing and securing their firmware is essential.

Embedded devices concerns were examined by Costin et al. (2014) and Cui and Stolfo (2010). A quantitative background on the number of vulnerabilities in embedded devices is explored in by Cui and Stolfo (2010). They applied the analysis to the various available embedded devices and found that over 144 countries had vulnerable devices, including private companies, government agencies, ISPs, and other entities (Cui & Stolfo, 2010). Other researchers focused on firmware images analysis. They collected a massive number of firmware images and used them to analyze the possible vulnerabilities in those devices. The number of firmware images they used was around 760 thousand files related to firmware images for the analysis, and they ended up using a sample of approximately 32 thousand files (Costin et al.,

2014). Although they discovered 32 new vulnerabilities in their research, they concluded that their study is insufficient, and more undiscovered vulnerabilities are present.

Chapter 3

Embedded Devices Analysis

In this section, we examine some of the embedded devices testing approaches. There are many available approaches for embedded devices testing, but we will examine only a few approaches that researchers used widely, including static and dynamic analysis approaches for testing embedded devices. We then examine some of the firmware analysis and testing tools developed in recent years.

Embedded Devices Analysis Methods

1. Static Analysis

Static analysis of a code generally examines the binary code of a given program in order to detect vulnerabilities. The static analysis approach had several limitations concerning embedded devices, including possibilities of information loss, changes due to compiler effects, variation in hardware architecture, and scalability (Qasem et al., 2021). However, static analysis is still applicable to embedded devices. A tool that dynamically compares functions of different programs that is useful to detect vulnerabilities is created by Egele et al. (2014). Other researchers combined other methods to apply static analysis effectively. They implemented a static analysis method with fuzzing and keywords detection to detect vulnerabilities in embedded devices (Chen et al., 2021; Salehi et al., 2020). The static analysis approach helped detect various vulnerabilities, but it still misses a lot more (Qasem et al., 2021).

2. Dynamic Analysis and Symbolic Execution

The dynamic analysis approach examines and monitors program behavior while the code runs (Qasem et al., 2021; Wright et al., 2021). Most firmware testing and fuzzing tools currently follow the dynamic analysis approach to test embedded devices' behavior and detect vulnerabilities (American Fuzzy Lop, n.d.; The AFL++ Fuzzing Framework, n.d.; Chen et al., 2019; Feng et al., 2020; Fowze et al., 2021; Huang et al., 2020; Mera et al., 2021; Qasem et al., 2021; Scharnowski et al., 2022; Wright et al., 2021; Zhou et al., 2021). Researchers' application of dynamic analysis followed several execution approaches, including concrete execution, symbolic execution, and concolic execution (Qasem et al., 2021; Wright et al., 2021). Concrete execution is where programs run over for specific user input, and the execution flow is then examined Baldoni et al. (2018). However, concrete execution had several limitations because it only explored one path for each round (Baldoni et al., 2018; Qasem et al., 2021).

Another dynamic analysis execution approach is symbolic execution, where programs take a symbol as an input instead of a concrete user value (Baldoni et al., 2018; Zhou et al., 2021). The primary advantage of this method is the ability to test several execution paths simultaneously. In addition, symbolic execution showed its effectiveness in detecting vulnerabilities in many programs (Baldoni et al., 2018). Nevertheless, this method still has several limitations (Baldoni et al., 2018; Qasem et al., 2021; Wright et al., 2021), but researchers overcame its challenges by combining concrete and symbolic execution leading to concolic execution (Baldoni et al., 2018; Zaddach et al., 2014).

The application of concolic execution in testing embedded devices was presented by Zhou et al. (2021), Zaddach et al. (2014), Böttinger and Eckert (2016), Cao (2020), and Dovgalyuk (2012). Their results showed the effectiveness of concolic execution in

discovering many vulnerabilities in embedded devices. For example, a few researchers presented the concolic execution method and its application in fuzzing, while others presented the effectiveness of concolic execution in emulating embedded devices' firmware behavior (Böttinger & Eckert, 2016; Cao et al., 2020). Concolic execution methods were implemented by Zhou et al. (2021) and Zaddach et al. (2014) in their firmware testing tools and managed to enhance the detection of vulnerabilities by combining concolic execution with other tools such as AFL fuzzer (*American Fuzzy Lop,* n.d.).

Concolic execution helped in developing tools with higher execution path coverage compared to concrete and symbolic execution. However, one limitation that current tools have is the ignorance of execution paths that do not trigger alarms of detected vulnerability. A few papers explored the limitations in firmware analysis and fuzzing tools. Their research presented that most traditional testing tools are not applicable to embedded devices because they ignore several memory corruptions errors (Muench et al., 2018; Wright et al., 2021). Thus, researchers are creating new tools to enhance the detection of vulnerabilities in new execution paths without missing some vulnerabilities. Instead of creating a new tool, our research will focus on analyzing all the possible execution paths to optimize the detection of vulnerabilities in the unexplored paths.

Embedded Devices Testing Methods

1. Traditional Fuzzing vs. Embedded Devices Fuzzing

Fuzz Testing, or Fuzzing, is a method used to test traditional PCs, and recently its usage in firmware testing has become popular. However, conventional fuzzing tools had issues related to memory corruption when applied to embedded devices. The main issue in fuzzing embedded devices is the ignorance of a few memory corruption errors that are not detected by traditional fuzzing tools. A few researchers analyzed the problem of memory corruption when performing fuzzing and highlighted six heuristics to improve fuzzing in firmware testing (Muench et al., 2018). They implemented those heuristics into two firmware testing tools and discovered that their heuristics were helpful and enhanced the fuzzing process. Other researchers followed different approaches to strengthen fuzzing and its application on embedded devices.

2. Embedded devices Fuzzing Tools

Embedded devices fuzzing tools vary in their methodology in performing fuzz testing, but their general purpose is generally detecting bugs in other software codes. Many researchers created fuzzing tools based on different algorithms (American Fuzzy Lop, n.d.; The AFL++ Fuzzing Framework, n.d.; Chen et al., 2018; Godefroid et al., 2017; Scharnowski et al., 2022). The American Fuzzy Lop (AFL) fuzzer is widely used by many security specialists to detect vulnerabilities in traditional PC programs generally, and it started involving embedded devices testing. However, the AFL fuzzer might require special configurations before running it over embedded devices' firmware. Other fuzzing tools could be based on a machine learning algorithm to create feeds for the program (Godefroid et al., 2017).

On the other hand, a few researchers designed fuzzing tools specifically for embedded devices. An app-based fuzzing method and a memory-mapped I/O (MMIO)-based fuzzing tools were implemented by Scharnowski et al. (2022) and Chen et al. (2018), respectively. Other researchers implemented fuzzers in their firmware analysis tools with other methods (Chen et al., 2019; Huang et al., 2020; Mera et al., 2021; Zaddach et al., 2014; Zhou et al., 2021). Section 3.3 explores a few fuzzing tools implemented in firmware testing tools. All the existing fuzzing tools serve the same purpose, vulnerabilities detection, but the difference lies in the algorithm and execution path generation. However, fuzzing tools researchers focused on the implementation of a new fuzzing algorithm, but they did not explore execution paths in the existing tools. For example, the AFL tool generates thousands of execution paths, but researchers explore only a few that trigger alerts.

3. Firmware testing tools

Although some researchers use fuzzing tools to test the firmware of embedded devices, many others implemented tools explicitly designed for firmware testing. Firmware testing tools mainly implement other methods, such as fuzzing, dynamic analysis, and machine learning (Cao et al., 2020; Fowze et al., 2021; Gustafson et al., 2019; Huang et al., 2020; Mera et al., 2021; Qasem et al., 2021; Wright et al., 2021; Zaddach et al., 2014; Zhou et al., 2021). Most firmware analysis researchers focused on emulating the behavior of embedded devices since testing them directly is not possible (Wright et al., 2021). The PRETENDER methodology was implemented by Gustafon et al. (2019) as the first proof-of-concept tool that emulates the behavior of embedded devices' firmware.

Other researchers created more tools that emulate and test firmware images of embedded devices. Various tools that emulate the behavior of embedded devices based on different algorithms were presented in many papers (Feng et al., 2020; Mera et al., 2021; Scharnowski et al., 2022; Zaddach et al., 2014; Zhou et al., 2021). A few researchers presented tools where the tool learns how the firmware behaves and emulates it to detect vulnerabilities (Feng et al., 2020; Zhou et al., 2021). Others implemented fuzzing tools in their firmware testing tools in order to enhance the emulation and path generation (Huang et al., 2020; Scharnowski et al., 2022; Zhou et al., 2021), while a few implemented new methods for firmware testing (Mera et al., 2021; Zaddach et al., 2014). Essentially, most firmware testing tools depend on executing and monitoring the behavior of the program. They focus on the execution paths where vulnerabilities might be present and ignore other execution paths.

Execution Paths

Research papers concerning execution paths of embedded devices have been limited only to a few articles. Most papers that follow the data analysis approach in analyzing execution paths are concerned with anomaly detection in the execution semantics of embedded devices. A few researchers created models to capture run-time execution paths and compare them to previously collected benign execution paths to detect abnormal executions (Cheng et al., 2017; Yoon et al., 2017). A tool called Orpheus is used to detect control-oriented, code-oriented, and data-oriented attacks by forcing execution semantics at run-time (Cheng et al., 2017). The tool forces embedded devices to follow execution semantics to avoid executing abnormal inputs by users.

Other researchers created a technique to detect the abnormal activities in execution paths but did not create a complete model for testing the detected abnormal executions (Yoon et a., 2017).

SyML is another tool used to collect and analyze execution paths (Ruaro et al., 2021). This tool utilizes a machine learning algorithm to detect execution paths and learn the typical patterns of embedded devices' behavior. It also monitors the execution of vulnerable programs and extracts certain features to allow the algorithm to learn how vulnerable programs behave and test other programs.

Chapter 4

Research Methodology

We are planning to survey some of the firmware testing tools that were developed recently and tested on embedded devices. The analysis we are going to conduct would use the same criteria used by Qasem et al. (2021) and Wright et al. (2021), but we are going to extend the analysis to include more essential factors. Before we proceed with the additional factors, we review the analysis conducted by both Qasem et al. (2021) and Wright et al. (2021).

1. Wright et al. (2021)

Wright et al. (2021) focused on the challenges in firmware re-hosting and emulation. In their paper, they examined the evolution of emulation techniques from the early 1970s until the current time and other vulnerability detection methods that contributed to the evolution of emulation, such as symbolic execution, concolic execution, and fuzzing. Wright et al. (2021) examined emulation techniques in detail with other factors that contributed to the effectiveness of the emulation, including the device being emulated, the purpose of the emulation, and the level of control over the emulated device. Essentially, they examined around 24 different papers that created tools or platforms for running the emulation. Although their work is not related to the analysis of firmware testing in embedded devices specifically, we believe that the criteria they used in the evaluation of firmware testing-related tools are essential for our research.

For instance, Wright et al. (2021) examined some of the challenges in emulation setup, including the availability of external hardware and peripherals, Mem interaction/ setup, hardware configuration, missing code, and function identification and labeling. We will use all the five criteria mentioned above and measure them on the surveyed papers in our work. Although Wright et al. (2021) examined other criteria, we consider them out of the scope of our analysis.

2. Qasem et al. (2021)

Qasem et al. (2021), on the other hand, focused on addressing the issues in automating the process of vulnerabilities detection in embedded devices. First, they examined the challenges faced by security practitioners when they apply the traditional binary analysis methods to embedded devices. Qasem et al. (2021) Highlighted fourteen critical challenges faced when applying binary analysis to any software, in which four of them are directly related to embedded devices, including hardware architecture, scalability, fault tolerance, and firmware reverse engineering. Qasem et al. (2021) also examined binary analysis approaches and their implications on embedded devices, including static analysis and dynamic analysis and its categories and symbolic execution. Before analyzing the surveyed papers, Qasem et al. (2021) classified embedded devices into three main categories: General purpose OS-based, embedded OS-based, and embedded devices without OS abstraction. Embedded devices tested were classified for each of the tools examined in their paper, with a summarized comparison between the tools.

One of the essential criteria Qasem et al. (2021) used in their analysis is the firmware analysis methodology that includes partial emulation, full emulation, symbolic execution, fuzzing, web interface checking, network scanning, taint analysis, and static analysis. We would also use the same classification in our analysis, but we would only keep the full emulation symbolic execution, fuzzing, and concolic execution since the other classifications are out of our scope. Other criteria researchers used for the classification is the CPU architecture, including the x86-64, the ARM, and the MIPS architectures. The same criteria would be used in our evaluation and comparison of the papers we would examine.

3. Our Approach

We would combine some of the criteria we mentioned earlier in both Wright et al. (2021) and Qasem et al. (2021) studies in our survey with additional factors. One of the factors we would focus on is the capability of the surveyed work to discover vulnerabilities in the dataset. Each of the surveyed tools has measured its effectiveness by discovering previously known vulnerabilities or new ones. In order to scale the effectiveness of each tool, we would evaluate the dataset used in each of the tools with the methods used to extract information from the firmware. Based on our evaluation of the dataset, we would measure the coverage of the dataset and tool capability on most of the embedded devices in the real world. The various types of embedded devices would make it almost impossible to generalize the results of a dataset, but we would try to cover most of the embedded devices we could find.

From the tools we chose to survey, not all of them managed to successfully detect and run the vulnerabilities on their sample. Thus, another factor we would use to evaluate the surveyed work is the success in executing their testing samples. Essentially, we would compare the results of each of the surveyed work and its limitations. The results would include an evaluation of each surveyed work individually and a comparison between each of the tools we surveyed. In the next section, we review the work done by each of the surveyed papers.

Chapter 5

Surveyed Work

In Chapter 3, we reviewed the methods that are mainly used to test embedded devices and enhance their security. We then reviewed a few papers that designed and explored firmware testing tools using different methods. We analyzed each of the tools to choose a few tools that we could survey and explore further. The following criteria were used for choosing the tools we would survey:

- One of the surveyed papers must include a fuzzer that explores various execution paths and generate input feed for the firmware testing process.
- One of the surveyed papers must include the capability to create models of the firmware images and help in the analysis. Such a tool might use AI or other components that would facilitate the process.
- Since we plan to extend on Qasem et al. (2021) and Wright et al. (2021) surveys, we must include at least one paper explored by one of the two papers to make our results comparable with theirs.
- Each of the tools must have explored firmware testing from a different perspective or using different methods. The only exception would be the paper that was surveyed previously by either Qasem et al. (2021) or Wright et al. (2021).

We evaluated 24 different papers that worked on improving the security of embedded devices and testing methods based on the criteria mentioned above. Essentially, we chose to survey four papers, including P2IM by Feng et al. (2021) (P2IM), DICE by Mera et al. 2021, and µEmu by Zhou et al. (2021), and HERA by Niesler et al. (2021). We chose P2IM was explored

and evaluated by Wright et al. (2021). Moreover, two of the other tools, µEmu, and DICE, were built on some of the ideas included in the P2IM paper. Thus, we decided to include this paper in our survey to make a comparison between the development of µEmu and DICE based on P2IM. On the other hand, both µEmu and DICE included fuzzing tools and different methods. The details of each of these tools are explored in the next section. Lastly, we included ProXray because it has the ability to model the protocols based on the firmware and facilitate vulnerabilities in embedded devices. We explore and review the work done by each of the aforementioned papers in the next sections.

P2IM

Process-Peripheral Interface Modeling (P2IM) is a firmware testing tool designed by Feng et al. (2021). This paper focuses on the dynamic testing, or what is called fuzzing, of firmware images. The title of the article is P2IM: Scalable and Hardware-independent Firmware Testing via. In this work, Feng et al. (2021) focused on implementing an analysis tool that would not require hardware dependencies to conduct firmware analysis for IoT devices. The research intended to create a program that will help in firmware testing for different IoT devices without requiring any specific hardware features. It also implemented fuzzing techniques that helped generate models for different peripherals. The research is guided by the issues related to the increase of using IoT devices in the industry and the need for more efficient methods for detecting vulnerabilities in IoT devices. A good hypothesis that would guide this research would be "implementing a fuzzer in generating peripherals' models would improve the process of firmware analysis without requiring hardware dependencies". As in the uEmu tool, described in the next

sections, P2IM has a similar logical variable to measure whether the program could detect vulnerabilities or not. Also, it includes other variables such as time. The main purpose of this research would be to measure the efficiency of the developed program, P2IM, in detecting vulnerabilities in different IoT devices.

Feng et al. (2021) conducted an experiment on a firmware imaged dataset, where they measured the effectiveness of the developed program by running firmware testing on different embedded devices. The designed methods for the P2IM were the main independent variables, and they would be applied in a similar way for all the firmware images in the used sample. The method used here is designing the P2IM program and implementing a fuzzer with other plug-ins to perform firmware testing on the firmware images in the sample. The researchers will collect the results for running the test and then measure the effectiveness of the program. Also, they wanted to know whether this new method, which requires no hardware dependencies, would be effective. The dataset explored by Feng et al. (2021) included a total of 70 firmware images, where 10 of them are from real-life firmware embedded devices, and 60 are testing firmware images used to evaluate the program only. The real-life embedded devices included a drone, a robot, and a Programmable Logic Controller (PLC_

Essentially, the paper succeeded in creating a firmware testing program, P2IM, that requires no hardware dependencies, and they managed to execute 79 percent of the used sample. The tool also managed to reveal some unique unknown bugs during the testing. Although P2IM managed to succeed in creating a hardware-independent testing tool, it still has many drawbacks that must be addressed. For instance, P2IM relies on the existing peripherals that were used during the development of the tool. If a new type of embedded device with a different peripheral was tested by P2IM, the tool would crash, and no results would be achieved.

µEmu

µEmu, an automated firmware emulation through an invalidity-guided knowledge interface tool designed by Zhou et al. (2021). It introduced a new method that emulates the CPUs of different embedded devices to perform firmware analysis of these devices. Unlike the P2IM firmware analysis tool that builds general models for peripherals, µEmu uses a symbolic execution method to extract information from the given firmware and learn about the different peripherals that might not be defined in a general model. It also executes the fuzzing tool. The main purpose of the uEmu program, similar to other firmware analysis programs, is to perform firmware testing for IoT devices and discover vulnerabilities. Zhou et al.'s (2021) paper was guided by the fact that the existing firmware analysis programs do not perform very well in discovering all the vulnerabilities, and encountering undefined peripheral in any firmware could lead the program to crash. The main purpose is to create a program that emulates the CPU of any firmware without knowing what kind of peripherals are expected.

µEmu is a firmware testing and analysis tool that incorporates various algorithms and tools to emulate and test embedded devices (Zhou et al., 2021). µEmu follows two phases in order to perform the testing. First, it follows the knowledge creation about the embedded device to learn its behavior. Second, it uses the knowledge to emulate the behavior of the embedded device and perform the testing. The µEmu tool implements three components in order to emulate the device and generate test cases:

S2E: S2E is an open-source symbolic execution platform that is based on QEMU that supports testing user applications and drivers (Chipounov et al., 2011). µEmu uses the S2E platform with KLEE to perform concolic execution on the embedded devices (Cadar et al., 2008; Zhou et al., 2021).

AFL: µEmu uses the AFL fuzzing tool to generate test cases based on the execution paths the program created during the knowledge creation phase (*American Fuzzy Lop*, n.d.). The test cases are then fed to the tool, and the tool will start creating logs based on the results of the execution.

KLEE: KLEE is another symbolic execution framework that µEmu implements with the S2E platform to optimize the results and perform concolic execution (Cadar et al., 2018). µEmu also uses KLEE to generate test cases, aside from AFL.

In Zhou et al.'s (2021) paper, they used many different variables to measure the effectiveness of the program that are similar to the ones used to evaluate P2IM in Feng et al.'s (2021) paper. Moreover, Zhou et al. (2021) used the same sample explored by Feng et al. (2021) to compare and evaluate the results of µEmu in conjunction with P2IM. Essentially, Zhou et al. (2021) evaluated the results of running firmware testing on the P2IM testing sample. They found that the program achieved a passing rate of 93 percent without manual assistance and 100 percent with manual assistance. Compared to the P2IM, uEmu achieved much better results. However, µEmu still has many drawbacks that must be addressed. First, µEmu failed to execute all the samples without human interaction because it uses fewer heuristics by leveraging symbolic execution, which would cause failure when a heuristic fails. Second, µEmu might not cover all the invalidity states because it fails at some points that the user must feed the program with before running it. Third, µEmu fails in detecting infinite loops if no symbol is used in that loop. In essence, µEmu contributed to improving firmware analysis in embedded devices, but it still has many drawbacks. One area that µEmu did not address in their research is the Direct Memory Access (DMA) in embedded devices. This area was further explored by Mera et al. (2021), who created the DICE plug-in described in the next section.

DICE

DMA Input Channel Emulation (DICE) is a plug-in component that could be built on existing or future firmware testing tools introduced by Mera et al. (2021). The main purpose of the DICE plug-in is to extend and give the ability for the firmware testing tool to emulate DMA input channels and generate DMA inputs without being hardware-dependent. When the DICE plug-in component is added to a firmware testing tool, it helps the tool in expanding its analysis to include firmware code, states, and vulnerabilities dependent on DMA input (Mera et al., 2021). Mera et al. (2021) work was driven by the fact that detecting bugs in microcontroller units (MCUs) that run the firmware of embedded devices is difficult, and many vulnerabilities might have been ignored due to ignorance of the content of the DMA. The various kinds of peripherals used by embedded devices make it very difficult to test them as the traditional PCs, and additional methods are required. Although other researchers have implemented methods that help in analyzing the DMA, such as μEmu by Zhou et al. (2021), they still cannot fully utilize the analysis of the DMA due to ignorance or the need for manual identification of DMA buffers (Mera et al., 2021).

Mera et al. (2021) explored the weaknesses of the previous firmware analysis and came up with the requirements needed in the DICE plug-in component. The requirements included hardware independency, firmware compatibility, dynamic DMA control, no source code needed, and easy integration with existing firmware analysis tools. Mera et al. (2021) integrated DICE drop-in component with the firmware analysis tool P2IM developed by Feng et al. (2021) that we discussed earlier and an emulator called PIC32. The dataset they tested included 83 benchmarks and sample firmware with nine different DMA controllers. Compared to previous firmware analysis tools that weren't able to emulate the DMA, DICE managed to detect 33 out of 37 DMA

input channels, and it managed correctly to supply 21 out of 22 DMA buffers that were used by firmware. In essence, DICE with P2IM detected five previously unknown vulnerabilities that P2IM alone was not able to detect. All the discovered bugs were tested, proved their presence, and showed that they could be exploited remotely (Mera et al., 2021).

As with the other tools, DICE has its weaknesses and limitations. First, the DICE plug-in is dependent on both the firmware analysis tool and the fuzzing tool. The limitations in the fuzzing tool, indeed, limit the performance of DICE. For instance, DICE would not be able to solve complex state machines and path constraints due to the lack of error detectors and sanitizers for MCUs. Second, some firmware uses a high amount of data in the DMA that drains the fuzzing tool input before going deeper in the code, limiting the ability to test them. Third, DICE made the assumption that something called transfer descriptor is always written in the DMA. However, transfer descriptors are sometimes written to the RAM instead of the DMA, which would not be explored in this case.

ProXray

ProXray is another firmware analysis tool introduced by Fowze et al. (2021). This tool focuses mainly on firmware analysis but not firmware testing. Indeed, ProXray focuses on understanding the behavior of the firmware and generating protocol models for firmware devices. Fowze et al. (2021) focused on the fact that embedded devices use a wide variety of complex peripherals and protocols connected to the firmware that allows for their functionality. The existence of the numerous protocols caused the lack of formal specifications that help in automating firmware analysis. Thus, the main purpose of the Fowze et al. (2021) paper is to

create a protocol model learning algorithm that helps in learning the behavior of known firmware of embedded devices and apply it to the unknown firmware of other embedded devices.

The tool followed several steps to perfume the automated analysis for the firmware of embedded devices. First, ProXray runs an algorithm to extract protocol models from the firmware that would be used as learning data. Second, ProXray starts learning and discovering the protocol fields and their functionalities from the extracted protocol models. Third, the tool performs symbolic execution on unknown firmware of embedded devices and explores the different paths, and detects the functionality of the protocol or the model. In essence, the tool will automate the process of analyzing the firmware of embedded devices.

In Fowze et al.'s (2021) paper, they executed ProXray over two known peripheral, USB and Bluetooth. The USB training dataset included 23 firmware images that implement three different classes of USB, while the testing dataset contained six firmware images. The Bluetooth training dataset included four firmware images, while the testing dataset included another four firmware images. In essence, the ProXray tool showed its effectiveness in extracting models from both the USB and Bluetooth firmware images. It managed to speed up the process of firmware analysis and automated it. One major limitation in this research is the weakness of the dataset. Although the tool might be applied to various embedded devices types, it was only tested on USB and Bluetooth. Although the nature of ProXray is different than the other tools we chose to survey, we decided to include it because we found it relevant to our analysis.

Chapter 6

A Comparison of The Surveyed Tools

After reviewing each of the surveyed works in chapter 5, we compare and evaluate each of the surveyed tools based on the criteria we specified in chapter 4. In this section, we will try to answer a few research questions that would help in understanding the surveyed work and the need for future tools. The following questions will be answered:

1. What are the essential and missing methodologies needed to perform firmware analysis and testing in the surveyed work?
2. What are the CPU architectures that are currently supported by the surveyed firmware testing and analysis tools, and what are their limitations?
3. What techniques must be supported by firmware analysis and testing tools to make the tool capable of detecting vulnerabilities efficiently?
4. To what extent the current firmware testing and analysis tools are able to detect vulnerabilities in embedded devices, and what do they lack?

Methodology

Table 1: Methodology Used by The Surveyed Work

Tool / Methodology	*Full Emulation*	*Symbolic Execution*	*Concolic Execution*	*Fuzzing*	*AI*
P2IM	Yes	No	No	Yes	No
µEmu	Yes	Yes	Yes	Yes	No
DICE	Yes	Yes	No	Yes	No
ProXray	N/A	Yes	N/A	N/A	Yes

As we can see in table 1, we compared the methodology used by each of the surveyed works. P2IM, µEmu, and DICE all support the full emulation of the firmware of the embedded device. P2IM was built to be one of the first emulators that are completely hardware-independent, so it was expected to have the ability to emulate the behavior of MCUs. P2IM did not support Symbolic execution, concolic execution, or AI, but it only supported fuzzing. Indeed, Feng et al. (2021) designed and tested the tool focusing on emulating MCUs in embedded devices, so we expected that their work would focus on emulation more than the other methodologies. Moreover, µEmu and DICE were developed after the creation of P2IM as superior tools that would address some of the issues mentioned in the P2IM tool. For instance, µEmu integrated full emulation, symbolic execution, concolic execution, and fuzzing by

employing various tools and platforms in its underlying structure. Although μEmu is one tool, it integrated various tools we mentioned in chapter 5, which led to its ability to apply different methodologies. On the other hand, DICE was only a drop-in component designed on top of existing firmware analysis tools. Indeed, it was built on P2IM when it was tested. Thus, we expected that DICE would have additional capabilities with all the capabilities of P2IM. DICE utilized full emulation, symbolic execution, and fuzzing. Although all three tools have the ability fully emulate the firmware behavior, their performance is different.

The last tool, ProXray, methodologies are distinct compared to the other tools. We expected this tool would behave in a completely different way since it focuses on firmware analysis only, not vulnerabilities detection. ProXray utilized AI on a testing sample of firmware images to learn about the behavior, characteristics, and functionality of a firmware image for a specific peripheral. The results of the protocol model learning were then used on testing cases that used similar peripherals that utilize symbolic execution.

To extend our analysis, we compared the tools we analyzed with the tools analyzed by Qasem et al. (2021), included in table 4, Appendix A. We can clearly see that most of the firmware analysis and testing tools utilized fuzzing. Moreover, the surveyed work by Qasem et al. (2021) is listed by the date of publication. The work listed at the bottom includes more methodologies used in the same tool. Comparing it to the tools we surveyed, the tools in our survey have utilized more methodologies than the older tools. Below, we summarize our findings for the methodologies of firmware testing and analysis tools:

Finding 1-1: None of the tools we analyzed or was analyzed by either Qasem et al. (2021) or Wright et al. (2021) utilized Artificial Intelligence (AI) to perform the analysis, leading to the need for human interaction.

Finding 1-2: Most of the tools are built to resolve a specific research problem, but they do not utilize other aspects explored by other tools.

Finding 1-3: Full emulation and fuzzing are some of the essential methodologies to perform firmware testing and analysis due to the lack of the ability to test embedded devices directly.

CPU Architecture

Another factor we would consider is the type of CPU architecture. As we can see in table 2 below, the CPU architectures supported by the surveyed tools vary depending on the purpose that the tool was designed for. For instance, P2IM was designed to emulate the behavior of embedded devices MCUs, which mostly use ARM-cortex as its CPU architecture. P2IM was not tested for either x86-64 or MIPS, so we are not sure if the tool supports the other CPU architectures. On the other hand, μEmu supports both x86-64 and ARM CPU architectures since it was included in the dataset that was tested. μEmu was not tested on MIPS architecture, so we are not sure it supports it. The other tool, DICE, could not be evaluated by itself since it is a plug-in that could be integrated into a firmware analysis tool. Thus, we included that DICE would support all the CPU architectures since it could easily be integrated into other firmware analysis tools. The last tool, ProXray, was used only on USB and Bluetooth devices. Those devices mainly run ARM and MIPS CPU architectures but not x86-64. Thus, ProXray supports both ARM and MIPS architectures. For the x86-64, we are not sure if ProXray would support it. Below, we summarize our findings for the CPU architecture of embedded devices supported by the surveyed tools.

Findings 2-1: The existing firmware testing and analysis tools cover the various CPU architectures available in today's embedded devices, but most of them only focus on a specific architecture.

Finding 2-2: One of the current limitations of firmware testing and tools is the fact that they lack the ability to cover CPU architectures that it was not designed for or were not tested on other architectures.

Table 2: CPU Architecture Support

Tool/ CPU Architecture	*x86-64*	*ARM*	*MIPS*
P2IM	No	Yes	No
µEmu	Yes	Yes	No
DICE	Yes*	Yes*	Yes*
ProXray	No	Yes	Yes

*DICE is a plug-in used with another firmware analysis and testing tool, so it would support the same firmware as the integrated firmware analysis tool.

Technique

Another comparison factor we would consider is the Techniques used by the surveyed work. Table 3 shows some of the techniques we used in the Wright et al. (2021) survey. We used the same techniques categories as Wright et al. (2021) work and extended it to include the tools we surveyed. First, external hardware and peripheral support are one of the techniques that most firmware analysis tools are trying to support. It means that the firmware analysis tools support various kinds of embedded devices to emulate and run the testing. Second, Memory interactions and setup support means that the tool would support locating and defining memory interactions.

Third, firmware analysis tools would also support hardware configuration, where it brings the hardware to a specific state that helps the emulating to occur. Fourth, most of the firmware images available for testing might contain missing code, so some firmware testing and analysis devices support firmware images with missing code. Fifth, function identification and labeling support mean that the firmware testing tool, or the emulating part of the tool, would support identifying the functions and labels used by the firmware to help security practitioners in understanding its behavior.

P2IM, µEmu, and DICE utilized the first four techniques in the comparison, including external hardware and peripheral support, memory interactions/ setup support, hardware configuration support, missing code support, and function identification and labeling support. On the other hand, ProXray, utilized external hardware and peripheral support, missing code, and function identification and labeling. Below, we summarize our findings for the various techniques used in the surveyed tools:

Finding 3-1: Most of the newly developed firmware testing and analysis tools support external hardware and peripherals, which help in covering and testing a wider range of embedded devices.

Finding 3-2: For emulating and testing the embedded devices' firmware, the tool would require memory interaction/ setup, hardware configuration, and missing code techniques that would help in covering a wider range of embedded devices and their execution paths during the testing and analysis.

Finding 3-3: Most newly developed firmware analysis and testing tools lack the function identification and labeling technique, while the older tools and traditional tools explored by Wright et al. (2021) mostly utilized this technique.

Table 3: Techniques Support

Tool / Technique	*External Hardware and Peripheral*	*Memory Interactions/ Setup*	*Hardware Configuration*	*Missing Code*	*Function Identification and Labeling*
P2IM	•	•	•	•	
µEmu	•	•	•	•	
DICE	•	•	•	•	
ProXray	•			•	•

Vulnerability Detection

Most of the tools we surveyed focused on vulnerability detection, except one, so we would also evaluate the ability of each tool to detect vulnerabilities. Although we plan to compare the capability of each tool, it would be difficult to make a direct comparison since each tool was designed to serve a specific purpose. P2IM and µEmu used the same dataset to test their performance. However, µEmu used an additional dataset of firmware samples that might not be testable by P2IM. For P2IM, the success of detecting the vulnerabilities in the dataset was 70 percent, while µEmu achieved a rate of 93 percent without manual assistance and 100 percent with manual assistance.

On the other hand, DICE, which was built on P2IM, managed to reveal five newly discovered vulnerabilities that P2IM did not. DICE used a different dataset sample compared to P2IM, but the sample used by it was not testable by P2IM. DICE showed that it could significantly improve the performance of P2IM, but it cannot be used as a standalone tool. The dataset used by DICE included both ARM-cortex and MIPS samples, but it did not include x86-64 firmware samples. Thus, comparing μEmu with P2IM and DICE, μEmu would have better capabilities than P2IM, but it is still not clear if μEmu outcomes DICE. Indeed, μEmu could be combined with DICE to achieve better vulnerability detection capability. Lastly, ProXray used a different dataset of USB and Bluetooth firmware samples. ProXray was not tested for vulnerability detection capability, but it could be used in conjunction with other firmware testing tools to improve the analysis of the firmware. The following points are the key findings from our analysis of vulnerabilities detection capabilities of firmware testing and analysis tools:

Finding 4-1: Each of the surveyed tools used a specific dataset that serves its purpose, so most of them are not generalizable, meaning they cannot be used for all the existing embedded devices or were not tested on them.

Finding 4-2: Two of the firmware testing tools managed to detect more vulnerabilities than the tool they were designed after and compared to, but they still lack some features that appeared in other tools we explored that could be integrated.

Finding 4-3: Firmware testing and analysis tools still lack the standardization capability, where each tool uses its syntax and features solely without being consistent with other tools.

In Appendix B, we summarize and list all the findings we found during our analysis for each of the subsections in chapter 3, including methodology, CPU architecture, technique, and vulnerability detection. In the next chapter, we address some of the issues we found during the comparison.

Chapter 7

Discussion

Based on our analysis of the factors in Qasem et al. (2021) and Wright et al. (2021) surveys, the datasets, the vulnerability detection capabilities, and the successfulness rate, we discovered a few key findings related to the development of firmware testing and analysis tools in the embedded devices industry. Aside from the key findings in chapter 6, we discuss some of the key issues in this section.

Lack of Real-time Vulnerability Patching

From the four tools we surveyed, three of them, P2IM, μEmu, and DICE, focused on detecting vulnerabilities in the firmware of the embedded device, while the last tool, ProXray, performed firmware analysis of firmware images sample. In other words, none of the tools have the capability to test and patch vulnerabilities while the tool is still running. Detecting vulnerabilities might be done on firmware images, but a patching process must be available to patch the detected bugs. Some devices might be patched by turning them off, but others are time-critical that must be patched while running in real-time. For instance, medical embedded devices that might cause high risk for their users must be addressed in real-time.

One study that conducted real-time patching for embedded devices we explored that implemented real-time patching capability was by Niesler et al. (2021). In this paper, Niesler et al. (2021) created a tool called HERA (Hot-patching of Embedded devices Applications) that performs in-time patching for embedded devices running on ARM-cortex. The tool succeeded in

patching bugs, but it lacks the ability to detect vulnerabilities by itself. Such a tool could be used in conjunction with a firmware testing and analysis tool like the ones we surveyed.

Lack of Coverage in Path Exploration

The three firmware testing tools we surveyed implemented a fuzzing tool as part of them. The fuzzing tool rule was to generate test cases, or possible execution paths, that the firmware testing tool would use to run the testing phase. In P2IM, the tool failed in detecting all the vulnerabilities but achieved a high success rate. When μEmu was designed, the μEmu achieved a higher success rate compared to P2IM. Moreover, DICE also managed to detect vulnerabilities that were not detectable by P2IM. We compared the tools based on the date of release with the tools explored by Qasem et al. (2021). We found that most of the tools rely on emulation and fuzzing tools in the testing process. Both P2IM and μEmu had a fuzzing tool implemented to generate test cases, but μEmu managed to achieve a higher success rate. We attribute the higher success rate to the ability of the tool to cover more execution paths that were not covered by the previous tool. Also, DICE managed to detect more vulnerabilities that were not detected by P2IM because P2IM did not cover test cases that extend to the DMA. Thus, we believe that further exploration of the possible execution paths in embedded devices is needed.

lack of scalability

Although most of the firmware analysis tools we mentioned succeeded in achieving their purpose, they all still lack the ability to perform large-scale firmware testing and analysis for embedded devices. The firmware testing process for each individual firmware image takes a lot

of time, and the testing sample is still small. There is no firmware testing tool so far that could be generalized to all the embedded devices. Moreover, testing a huge number of embedded devices would be very expensive.

Conclusion

In conclusion, embedded devices started involving in the industry, where a huge number of them are being used currently. Embedded devices' involvement caused many security threats and concerns for organizations, leading to the use of firmware testing tools to detect vulnerabilities in them. In this paper, we first reviewed some of the concerns of embedded devices. Second, we reviewed some of the essential methods and tools used to conduct firmware testing for embedded devices. Third, we described our approach for the analysis that extends from Qasem et al. (2021) and Wright et al. (2021) surveys. Fourth, we described the criteria we used to choose the tools we would survey and reviewed the work of each of them individually. Fifth, we made a cross-comparison between the tools we chose to survey. Finally, we concluded the paper with a discussion of our key findings. Our key findings show that embedded devices suffer from the lack of real-time patching techniques, lack of coverage in path exploration, and lack of scalability.

Appendix A

Tables and Figures used from other Papers

Table 4: Methodology Comparison by Qasem et al. (2021).

PROPOSAL	VENUE	Methodology							
		Partial Emulation	Full Emulation	Symbolic Execution	Fuzzing	Web Interface Check	Network Scanning	Taint Analysis	Static Analysis
Koscher et al. [93]	S&P 2010				•				
Cui et al. [39]	ACSAC 2010				•		•		
Mulliner et al. [110]	USENIX 2011				•				
Heninger et al. [77]	USENIX 2012				•		•		
Kamel et al. [86]	IJINS 2013				•				
FIE [48]	USENIX 2013			•					
RPFUZZER [147]	TIIS 2013				•				
Almgren et al. [4]	CRISALIS 2014				•				
Van et al. [143]	ESSoS 2014				•				
PROSPECT [89]	ASIACCS 2014	•							
AVATAR [154]	NDSS 2014	•							
Alimi et al. [3]	HPCS 2014		•		•				
Chen et al. [30]	TR 2014			•					
Lee et al. [97]	AINA 2015				•				
FIRMALICE [135]	NDSS 2015			•					•
SURROGATES [94]	WOOT 2015	•							
FIRMADYNE [28]	NDSS 2016		•			•			
Kammerstetter et al. [88]	SECUWARE 2016	•							
Costin et al. [37]	ASIACCS 2016		•						
AVATAR^2 [108]	BAR 2018	•							
INCEPTION [36]	USENIX 2018			•					
IoTFUZZER [29]	NDSS 2018				•			•	
IoTHunter [152]	CCS 2019		•		•				
WMIFUZZER [144]	SCN 2019				•	•			
FIRM-AFL [158]	USENIX 2019	•	•		•				
PRETENDER [73]	RAID 2019		•						
Ai et al. [2]	ICCSP 2020			•					
DISTRIBUTION	NA.	22%	22%	15%	52%	7%	7%	4%	4%

Appendix B

A Summary of The Findings

Finding 1-1: None of the tools we analyzed or was analyzed by either Qasem et al. (2021) or Wright et al. (2021) utilized Artificial Intelligence (AI) to perform the analysis, leading to the need for human interaction.

Finding 1-2: Most of the tools are built to resolve a specific research problem, but they do not utilize other aspects explored by other tools.

Finding 1-3: Full emulation and fuzzing are some of the essential methodologies to perform firmware testing and analysis due to the lack of the ability to test embedded devices directly.

Findings 2-1: The existing firmware testing and analysis tools cover the various CPU architectures available in today's embedded devices, but most of them only focus on a specific architecture.

Finding 2-2: One of the current limitations of firmware testing and tools is the fact that they lack the ability to cover CPU architectures that it was not designed for or were not tested on other architectures.

Finding 3-1: Most of the newly developed firmware testing and analysis tools support external hardware and peripherals, which help in covering and testing a wider range of embedded devices.

Finding 3-2: For emulating and testing the embedded devices' firmware, the tool would require memory interaction/ setup, hardware configuration, and missing code techniques that would help in covering a wider range of embedded devices and their execution paths during the testing and analysis.

Finding 3-3: Most newly developed firmware analysis and testing tools lack the function identification and labeling technique, while the older tools and traditional tools explored by Wright et al. (2021) mostly utilized this technique.

Finding 4-2: Two of the firmware testing tools managed to detect more vulnerabilities than the tool they were designed after and compared to, but they still lack some features that appeared in other tools we explored that could be integrated.

Finding 4-3: Firmware testing and analysis tools still lack the standardization capability, where each tool uses its syntax and features solely without being consistent with other tools.

BIBLIOGRAPHY

Abbasi, A., Wetzels, J., Holz, T., & Etalle, S. (2019). Challenges in Designing Exploit Mitigations for Deeply Embedded Systems. 2019 IEEE European Symposium on Security and Privacy (EuroS P), 31–46. https://doi.org/10.1109/EuroSP.2019.00013

Ai, C., Dong, W., & Gao, Z. (2020). A Novel Concolic Execution Approach on Embedded Device. Proceedings of the 2020 4th International Conference on Cryptography, Security and Privacy, 47–52. https://doi.org/10.1145/3377644.3377654

Almakhdhub, N. S., Clements, A. A., Payer, M., & Bagchi, S. (2019). BenchIoT: A Security Benchmark for the Internet of Things. 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 234–246. https://doi.org/10.1109/DSN.2019.00035

American Fuzzy Lop. (n.d.). Retrieved November 4, 2021, from https://lcamtuf.coredump.cx/afl/

Antonakakis, M., April, T., Bailey, M., Bernhard, M., Bursztein, E., Cochran, J., Durumeric, Z., Halderman, J. A., Invernizzi, L., Kallitsis, M., Kumar, D., Lever, C., Ma, Z., Mason, J., Menscher, D., Seaman, C., Sullivan, N., Thomas, K., & Zhou, Y. (2017). Understanding the Mirai Botnet. 1093–1110. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis

Baldoni, R., Coppa, E., D'elia, D. C., Demetrescu, C., & Finocchi, I. (2018). A Survey of Symbolic Execution Techniques. ACM Computing Surveys, 51(3), 50:1-50:39. https://doi.org/10.1145/3182657

Böttinger, K., & Eckert, C. (2016). DeepFuzz: Triggering Vulnerabilities Deeply Hidden in Binaries. In J. Caballero, U. Zurutuza, & R. J. Rodríguez (Eds.), Detection of Intrusions and Malware, and Vulnerability Assessment (pp. 25–34). Springer International Publishing. https://doi.org/10.1007/978-3-319-40667-1_2

Brooks, T. N. (2018). Survey of Automated Vulnerability Detection and Exploit Generation Techniques in Cyber Reasoning Systems. ArXiv:1702.06162 [Cs]. http://arxiv.org/abs/1702.06162

Cadar, C., Dunbar, D., & Engler, D. (2008). KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, 209–224.

Cao, C., Guan, L., Ming, J., & Liu, P. (2020). Device-agnostic Firmware Execution is Possible: A Concolic Execution Approach for Peripheral Emulation. Annual Computer Security Applications Conference, 746–759. https://doi.org/10.1145/3427228.3427280

Chen, J., Diao, W., Zhao, Q., Zuo, C., Lin, Z., Wang, X., Lau, W. C., Sun, M., Yang, R., & Zhang, K. (2018). IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. Proceedings 2018 Network and Distributed System Security Symposium. Network and Distributed System Security Symposium, San Diego, CA. https://doi.org/10.14722/ndss.2018.23159

Chen, L., Wang, Y., Cai, Q., Zhan, Y., Hu, H., Linghu, J., Hou, Q., Zhang, C., Duan, H., & Xue, Z. (2021). Sharing More and Checking Less: Leveraging Common Input Keywords to Detect Bugs in Embedded Systems. 303–319. https://www.usenix.org/conference/usenixsecurity21/presentation/chen-libo

Chen, T., Zhang, X.-S., Ji, X.-L., Zhu, C., Bai, Y., & Wu, Y. (2015). Test Generation for Embedded Executables via Concolic Execution in a Real Environment. IEEE Transactions on Reliability, 64(1), 284–296. https://doi.org/10.1109/TR.2014.2363153

Chen, Y., Li, P., Xu, J., Guo, S., Zhou, R., Zhang, Y., Taowei, & Lu, L. (2019). SAVIOR: Towards Bug-Driven Hybrid Testing. ArXiv:1906.07327 [Cs]. http://arxiv.org/abs/1906.07327

Cheng, L., Tian, K., & Yao, D. (Daphne). (2017). Orpheus: Enforcing Cyber-Physical Execution Semantics to Defend Against Data-Oriented Attacks. Proceedings of the 33rd Annual Computer Security Applications Conference, 315–326. https://doi.org/10.1145/3134600.3134640

Chipounov, V., Kuznetsov, V., & Candea, G. (2011). S2E: A platform for in-vivo multi-path analysis of software systems. Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, 265–278. https://doi.org/10.1145/1950365.1950396

Clements, A. A., Gustafson, E., Scharnowski, T., Grosen, P., Fritz, D., Kruegel, C., Vigna, G., Bagchi, S., & Payer, M. (2020). HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. 1201–1218. https://www.usenix.org/conference/usenixsecurity20/presentation/clements

Corteggiani, N., Camurati, G., & Francillon, A. (2018). Inception: System-Wide Security Testing of Real-World Embedded Systems Software. 309–326. https://www.usenix.org/conference/usenixsecurity18/presentation/corteggiani

Costin, A., Zaddach, J., Francillon, A., & Balzarotti, D. (2014). A Large-Scale Analysis of the Security of Embedded Firmwares. 95–110. https://www.usenix.org/node/184450.

Costin, A., Zarras, A., & Francillon, A. (2016). Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces. Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, 437–448. https://doi.org/10.1145/2897845.2897900

Cui, A., Costello, M., & Stolfo, S. (2013). When Firmware Modifications Attack: A Case Study of Embedded Exploitation. https://doi.org/10.7916/D8P55NKB

Cui, A., & Stolfo, S. J. (2010). A quantitative analysis of the insecurity of embedded network devices: Results of a wide-area scan. Proceedings of the 26th Annual Computer Security Applications Conference, 97–106. https://doi.org/10.1145/1920261.1920276

David, Y., Partush, N., & Yahav, E. (2017). Similarity of binaries through re-optimization. Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, 79–94. https://doi.org/10.1145/3062341.3062387

David, Y., Partush, N., & Yahav, E. (2018). FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware. ACM SIGPLAN Notices, 53(2), 392–404. https://doi.org/10.1145/3296957.3177157

Dovgalyuk, P. (2012). Deterministic Replay of System's Execution with Multi-target QEMU Simulator for Dynamic Analysis and Reverse Debugging. 2012 16th European Conference on Software Maintenance and Reengineering, 553–556. https://doi.org/10.1109/CSMR.2012.74

Egele, M., Woo, M., Chapman, P., & Brumley, D. (2014). Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components. 303. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/egele

Feist, J., Mounier, L., Bardin, S., David, R., & Potet, M.-L. (2016). Finding the needle in the heap: Combining static analysis and dynamic symbolic execution to trigger use-after-free. Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering, 1–12. https://doi.org/10.1145/3015135.3015137

Feng, B., Mera, A., & Lu, L. (2020). P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. 1237–1254. https://www.usenix.org/conference/usenixsecurity20/presentation/feng

Fowze, F., Tian, D., Hernandez, G., Butler, K., & Yavuz, T. (2021). ProXray: Protocol Model Learning and Guided Firmware Analysis. IEEE Transactions on Software Engineering, 47(9), 1907–1928. https://doi.org/10.1109/TSE.2019.2939526

Godefroid, P., Peleg, H., & Singh, R. (2017). Learn&Fuzz: Machine Learning for Input Fuzzing. ArXiv:1701.07232 [Cs]. http://arxiv.org/abs/1701.07232

Gustafson, E., Muench, M., Spensky, C., Redini, N., Machiry, A., Fratantonio, Y., Balzarotti, D., Francillon, A., Choe, Y. R., Kruegel, C., & Vigna, G. (2019). Toward the Analysis of Embedded Firmware through Automated Re-hosting. 135–150. https://www.usenix.org/conference/raid2019/presentation/gustafson

Huang, H., Yao, P., Wu, R., Shi, Q., & Zhang, C. (2020). Pangolin: Incremental Hybrid Fuzzing with Polyhedral Path Abstraction. 2020 IEEE Symposium on Security and Privacy (SP), 1613–1627. https://doi.org/10.1109/SP40000.2020.00063

Mera, A., Feng, B., Lu, L., Kirda, E., & Robertson, W. (2021). DICE: Automatic Emulation of DMA Input Channels for Dynamic Firmware Analysis. 17.

Muench, M., Stijohann, J., Kargl, F., Francillon, A., & Balzarotti, D. (2018). What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. Proceedings 2018

Network and Distributed System Security Symposium. Network and Distributed System Security Symposium, San Diego, CA. https://doi.org/10.14722/ndss.2018.23166

Niesler, C., Surminski, S., & Davi, L. (2021). HERA: Hotpatching of Embedded Real-time Applications. NDSS. https://doi.org/10.14722/NDSS.2021.24159

Oser, P., Feger, S., Woźniak, P. W., Karolus, J., Spagnuelo, D., Gupta, A., Lüders, S., Schmidt, A., & Kargl, F. (2020). SAFER: Development and Evaluation of an IoT Device Risk Assessment Framework in a Multinational Organization. Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies, 4(3), 114:1-114:22. https://doi.org/10.1145/3414173

Qasem, A., Shirani, P., Debbabi, M., Wang, L., Lebel, B., & Agba, B. L. (2021). Automatic Vulnerability Detection in Embedded Devices and Firmware: Survey and Layered Taxonomies. ACM Computing Surveys, 54(2), 25:1-25:42. https://doi.org/10.1145/3432893

Reaper: The Next Evolution of IoT Botnets. (2017, November 16). Fortinet Blog. https://www.fortinet.com/blog/threat-research/reaper-the-next-evolution-of-iot-botnets.html

Ruaro, N., Zeng, K., Dresel, L., Polino, M., Bao, T., Continella, A., Zanero, S., Kruegel, C., & Vigna, G. (2021). SyML: Guiding Symbolic Execution Toward Vulnerable States Through Pattern Learning. 24th International Symposium on Research in Attacks, Intrusions and Defenses, 456–468. https://doi.org/10.1145/3471621.3471865

Salehi, M., Hughes, D., & Crispo, B. (2020). μSBS: Static Binary Sanitization of Bare-metal Embedded Devices for Fault Observability. 381–395. https://www.usenix.org/conference/raid2020/presentation/salehi

Scharnowski, T., Bars, N., Schloegel, M., Gustafson, E., Muench, M., Vigna, G., Kruegel, C., Holz, T., & Abbasi, A. (2022). Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing. 18.

The AFL++ fuzzing framework. (n.d.). AFLplusplus. Retrieved November 5, 2021, from https://aflplus.plus/

Wright, C., Moeglein, W. A., Bagchi, S., Kulkarni, M., & Clements, A. A. (2021). Challenges in Firmware Re-Hosting, Emulation, and Analysis. ACM Computing Surveys, 54(1), 5:1-5:36. https://doi.org/10.1145/3423167

Yoon, M.-K., Mohan, S., Choi, J., Christodorescu, M., & Sha, L. (2017). Learning Execution Contexts from System Call Distribution for Anomaly Detection in Smart Embedded System. Proceedings of the Second International Conference on Internet-of-Things Design and Implementation, 191–196. https://doi.org/10.1145/3054977.3054999

Zaddach, J., Bruno, L., & Balzarotti, D. (2014). AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In Network and Distributed System Security Symp.

Zhou, W., Guan, L., Liu, P., & Zhang, Y. (2021). Automatic Firmware Emulation through Invalidity-guided Knowledge Inference. 19.

ACADEMIC VITA

Raed Katib
E-mail Address: rfk5242@psu.edu

EDUCATION

Bachelor of Science in Information Sciences and Technology
The Pennsylvania State University, Schreyer Honors College
Minor in Security and Risk Analysis

SKILLS

- Programming and scripting: Java, Bash, R, HTML
- Operating systems: Linux (Kali Linux, Ubuntu), Windows, MacOS
- Project management and design: MS Visio, MS Project, MS Office
- Other: Wireshark, Nmap, MySQL, Photoshop

ACADEMIC EXPERIENCE

Cyber-Security and Risk Analysis — Aug 2020 – Dec 2020

- Detected security vulnerabilities via SQL injection against a Mutillidae OWASP web application.
- Captured and analyzed transmitted packets on the network via Wireshark to identify suspicious activities.
- Utilized Nmap to recognize open and unsecured ports within firewalls.
- Understanding of access control, types of cyber-attacks, cryptography (public-key encryption, AES encryption), authentication threats, and incident response policies.
- Understanding of cyber-attacks process, and the counter measures applied to reduce or mitigate risks.

Database Project Management — Jan 2020 – May 2020

- Experience with restructuring database tables to eliminate redundancy.
- Knowledgeable in mining knowledge from relational databases.
- Involved in redesigning schemas and relations to improve accuracy and processing.
- Triaged issues, assessed goals and deadlines assuming the role of a project manager.

Data Mining and Big Data — Aug 2020 – Dec 2020

- Utilized R and machine learning models to predict cases of diabetes in a population.
- Applied data mining techniques (e.g., regression, decision trees, SVM) to confirm results published in healthcare journals.

Business and Administration — Aug 2020 – Feb 2021

- Familiarity with business concepts and entities (e.g., supply chain, digital business, ERP, EA).
- Analyzed the Boeing 737 Max issue and identified its causes and business impact.
- Familiarity with accounting and financial terms (e.g., ROI, NPV, cash flow).
- Understanding of UML elements, and used it in creating use cases and activity diagrams.

HONORS AND ACTIVITIES

Completed Certified Ethical Hacker (CEHv.10) through self-study.

- Used different security tools (e.g., Metasploit, Nmap) — 2021

Dean's List Recipient for outstanding academic performance (All Semesters) — 2020
Attended ICND1 and ICND2 training for Cisco. — 2019

- Performed packet analysis in each (TCP/IP) layer of the network.
- Experience with port scanning, and packet recording