XPLICIT: STATIC INFORMATION FLOW ANALYSIS FOR ARM32 FIRMWARE

A Dissertation
Presented to
The Academic Faculty

By

Konstantinos Karakatsanis

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in Electrical and Computer Engineering
School of Electrical and Computer Engineering

Georgia Institute of Technology

August 2024

© Konstantinos Karakatsanis 2024

XPLICIT: STATIC INFORMATION FLOW ANALYSIS FOR ARM32 FIRMWARE

Thesis committee:

Dr. Angelos D. Keromytis
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Newman Fabian Monrose
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Mustaque Ahamad
School of Cybersecurity and Privacy
Georgia Institute of Technology

Date approved: July 19, 2024

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

CFG Control Flow Graph

CVE Common Vulnerabilities and Exposures

DFG Data Flow Graph

DuT Device under Test

ICS Industrial Control System

IoT Internet of Things

PDU Protocol Data Unit

SVD System View Descriptor

VTOR Vector Table Offset Register

SUMMARY

In this work, we designed and implemented Xplicit; a static approach that aims to help identify potential vulnerabilities in firmware. The approach performs inter-procedural information flow analysis to track if untrusted data coming from different sources can reach sinks of interest after propagation. Our method takes into account two important elements, namely (1) implicit data flows and (2) the access of hardware control registers.

We leveraged IDA Pro to disassemble firmware binaries. Then, we visualized the information flows and the corresponding instructions using NetworkX graphs. Finally, we scaled the analysis by parallelizing it with GNU Parallel and running IDA Pro in autonomous mode. Our approach is the first implementation to identify implicit data flows in ARM32 firmware binaries to the best of our knowledge. In addition, it minimizes the dependency on IDA Pro (after disassembly), so even less technical people or people with no IDA Pro knowledge will be enabled to look at the information flows and detect potential vulnerabilities.

Our research can have a huge impact because it could identify potential vulnerabilities affecting devices running ARM32 firmware. Such devices can be found everywhere, from home Internet of Things (IoT) devices used by individuals to field devices used by an Industrial Control System (ICS).

CHAPTER 1
INTRODUCTION

The Internet of Things (IoT) refers to the network of devices that leverage internet connectivity to enhance their functionality, such as being controlled remotely or exchanging information with other systems and devices. These interconnected devices, known as IoT devices, range from household appliances to industrial machines, and their presence is rapidly expanding across domains. Unfortunately, like any computer system, IoT devices are consistently found to be vulnerable to both known and unknown security threats. For example, Zhao et al. [1] identified 385,060 IoT devices as vulnerable to the N-days vulnerability attack, and Ripple 20 project [2] discovered 19 0-day vulnerabilities that could affect hundreds of millions of devices. When these vulnerabilities are not patched, they leave IoT devices exposed to growing and increasingly sophisticated malware attacks, posing significant threats to user privacy, data security, and even physical safety. In fact, a recent report by Zscaler indicates a significant rise of 400% in IoT and OT malware attacks compared to the previous year [3].

Given the alarming trend of increased attacks on IoT devices, it is imperative to proactively identify and address their vulnerabilities. As an IoT device consists of hardware and firmware, one popular way of uncovering vulnerabilities is through firmware analysis. This is especially important as firmware is the foundational software that directly interfaces with the hardware, representing a critical attack surface where security flaws can have far-reaching consequences. Firmware analysis can be achieved through three main methods: static analysis, dynamic analysis, and symbolic execution. Static analysis involves inspecting the firmware code without executing the firmware binary. This method allows us to examine the code for potential vulnerabilities without the need to run on a device or emulator. On the contrary, dynamic analysis involves executing the firmware binary to observe

its behavior in real-time. Lastly, symbolic execution pertains to the analysis of firmware without executing it but instead, representing it as an abstraction and using mathematical expressions (symbols) to explore the feasible paths of execution.

Prior research has advanced firmware analysis and vulnerability detection through tools like BootStomp [4] and FirmXRay [5]. Unfortunately, these tools have limitations: BootStomp seems to not work anymore and relies on outdated software, while FirmXRay has a limited scope, targeting only Bluetooth link layer vulnerabilities on TI and Nordic devices. Additionally, many existing works focus on basic static analyses, such as string extraction and code matching to known vulnerabilities [6, 7, 8, 9, 1], often missing more complex issues like implicit data flows. Our research addresses these gaps by integrating both explicit and implicit data flow analyses, providing a comprehensive and scalable approach to firmware analysis.

1.1 Thesis Proposition

The objective of our research is to design and implement a static approach that aids in identifying potential vulnerabilities in firmware by employing inter-procedural information flow analysis to track whether untrusted data from various sources can reach the hardware control registers (sinks) after propagation, taking into account both data flows that reach these registers and implicit data flows.

1.2 Contributions

To the best of our knowledge, we are the first to develop a method that identifies implicit data flows in ARM32 firmware binaries. Additionally, it minimizes the dependency on IDA Pro after disassembly, enabling even less technical individuals or those without IDA Pro knowledge to analyze the information flows and detect potential vulnerabilities. Using our approach with IDA Pro and custom IDAPython scripts, we were able to reverse-engineer 8 IoT device binaries and identified one potential vulnerability.

Additionally, we are the first to provide a framework that aids analysts in separating the boot state from the steady state, enabling the detection of uninitialized resource (e.g., peripherals) usage. Although this last analysis still requires manual effort, it represents a significant step towards more comprehensive and automated firmware vulnerability detection.

Our research can have a significant impact because it could be used as a stepping stone/-first step to identify potential vulnerabilities affecting devices running ARM32 firmware. Such devices can be found everywhere, from home IoT devices used by individuals to field devices used by an Industrial Control System (ICS).

1.3 Thesis Overview

This thesis is structured as follows: We begin with a brief background in chapter 2, where we discuss various types of firmware for IoT devices, static analysis of firmware, implicit data flows, hardware control registers, and vulnerabilities we expect to detect with Xplicit. In chapter 3, we review related work, including firmware collection techniques and analysis methodologies. This chapter also compares existing tools. In chapter 4 we detail our methodology, including an overview of our infrastructure, prototype development, analysis at scale, and various techniques such as control flow analysis, data flow analysis, and enhanced data flow analysis. We also discuss our implicit data flow analysis in this chapter. In chapter 5 we present the results of our research, covering prototype development results, identifying risky flow in firmware, and analysis at scale results. In chapter 6, we discuss the limitations of our approach and propose future work to enhance our analysis techniques. Finally, chapter 7 provides the conclusion of our research.

CHAPTER 2
BACKGROUND

2.1 Types of Firmware for IoT Devices

There are two main categories of firmware for IoT devices. The first is bare-metal firmware, which consists of a single binary running directly on the device's hardware. In literature, this type of firmware is also referred to as monolithic, single-binary, or blob firmware. The second is OS-like firmware, which consists of multiple binaries and usually has more complex functionality.

2.2 Static Analysis of Firmware

To conduct static analysis, we must first obtain the source code of the firmware, which is a difficult task, as we discuss later in chapter 3. An alternative approach involves analyzing the assembly code, which is readily obtained through binary disassembly. The assembly code serves as a foundation for static information flow analysis (also known as static data flow analysis), which facilitates the tracing of how information propagates from a source (e.g., input from a peripheral) to a sink (e.g., hardware control register). This type of analysis is also referred to as static taint analysis because it tracks how data originating from sources of interest, known as tainted data, propagates to determine if and how it reaches designated sinks within the program.

2.3 Implicit Data Flows

To illustrate implicit data flows and how they differentiate from (explicit) data flows, we list a small pseudocode snippet in Listing 2.1. In line 1, variable x is defined. In line 2, variable y is calculated based on x; thus, y depends directly on x, indicating an information

flow from x to y. However, variable z appears to be independent of x since it is assigned a hardcoded value. In reality, though, z is implicitly dependent on x, because its value is determined by the condition of x being greater than 0. If $x > 0$, z is set to 1; otherwise, z is set to 0. This demonstrates why we classify this type of information flow as implicit.

```
x = 1
y = x - 1
if x > 0:
    z = 1
else:
    z = 0
```

Listing 2.1: Implicit Data Flow Example

2.4 Hardware Control Registers

Another overlooked part of the firmware are the hardware control registers. Only recently have researchers begun to examine these more closely for improved emulation and testing of firmware [10]. This effort, led by Feng et al., aimed to enhance dynamic analysis by more accurately emulating firmware peripherals. To accomplish this, they utilized the models they had developed concerning the operation of peripherals, focusing on their associated hardware control registers, access patterns, and their management. As a result, their system successfully identified seven previously unknown vulnerabilities in ten real-world firmware samples.

2.5 Vulnerabilities of Interest

Before we continue with the rest of the thesis, we believe it is essential to elaborate on the vulnerabilities that our tool will be able to identify.

A vulnerability that we expect to detect is the taint-style vulnerability. We expect to identify cases where input from untrusted or tainted sources influences critical operations in the firmware through implicit data flow analysis. This analysis could potentially expose injection attacks or data corruption risks. To the best of our knowledge, our tool is the first

to discover implicit data flows in firmware, suggesting that prior work may not have had the capability to detect such vulnerabilities through this method.

We have already identified a potential vulnerability of this type, where the value held by the control register UART5_SR (source), which depends on the Modbus protocol, impacts the hardware control register USB_EP0R (sink). This could cause unpredictable behavior and potentially disrupt the USB functionality. We present our findings in detail in the subsection Identifying Risky Flow in Firmware.

Another vulnerability that we expect Xplicit to help identify is the use of an uninitialized resource. As we mentioned earlier, by separating the boot state and the steady state of firmware, we aim to detect such cases by comparing the hardware control registers that are initialized during boot with those used during the steady operation of the DuT. To the best of our knowledge, no other work has attempted to separate the boot state from the steady state of firmware, leaving such vulnerabilities undetected by prior research.

CHAPTER 3
RELATED WORK

The literature survey is divided into 3 sections: Firmware Collection, Firmware Analysis, and Evaluation & Comparison of Existing Tools.

Firmware Collection summarizes existing techniques that are used for the collection of firmware. Firmware Analysis elaborates on the techniques that have been used for the analysis of firmware. Finally, we conclude in Evaluation & Comparison of Existing Tools with an overview of previous works' comparison.

3.1 Firmware Collection

Obtaining firmware samples is not trivial, as most vendors do not publish binaries, which makes it difficult for researchers to analyze them. Prior research has commonly employed crawling as the primary technique to obtain firmware. Researchers extensively utilized crawling methods to access vendor websites and support pages [6, 11, 9, 12, 13], FTP sites [8], and Google Play [5]. Other less popular techniques that have been used include extraction of the firmware image from the corresponding mobile Apps [5, 14], and a phantom device [15]. Wen et al. [5] specifically crawled Google Play to download the mobile Apps and extract the firmware image from them. Additionally, Zhu et al. [14] tried to use firmware dumps exported by the ports used for debugging (UART or JTAG) but found that the manufacturers had disabled the debugging ports.

3.2 Firmware Analysis

Ensuring the security of embedded systems necessitates firmware analysis and robust firmware vulnerability detection techniques. Researchers have explored various approaches to ana-

lyze firmware and identify weaknesses in firmware binaries. This section delves into the primary methods employed in the field, categorized into static analysis, dynamic analysis, and symbolic execution.

3.2.1 Static Analysis

Static analysis techniques examine the firmware code without executing it, allowing for rapid analysis of large numbers of firmware images. The primary static analysis techniques used for firmware vulnerability detection include:

- **Static Information Flow Analysis:** These methods track data flow within the firmware to identify potential security vulnerabilities, such as information leaks or unauthorized privilege escalation. Redini et al. [4] used BootStomp for both static and dynamic symbolic execution to uncover vulnerabilities in bootloaders. Redini et al. [16] developed Karonte, a framework designed for the static analysis of multi-binary firmware to detect vulnerabilities arising from complex binary interactions. Cheng et al. [12] proposed DTaint, a tool for static binary analysis focusing on detecting taint-style vulnerabilities in Linux-based firmware. Wen et al. [5] introduced FirmXRay, which leverages static analysis to detect Bluetooth link layer vulnerabilities in firmware.

- **Code Matching:** This approach compares firmware code to databases of known vulnerable code snippets and analyzes firmware versions to identify potential security vulnerabilities. For instance, Costin et al. [6] conducted large-scale static analysis of firmware images, highlighting the prevalence of known vulnerabilities in numerous devices. Similarly, David et al. [9] developed FirmUp, a tool designed for the static detection of known vulnerabilities by matching firmware code against known CVE.

- **Feature Detection:** This technique involves identifying patterns and characteristics within the firmware code that might indicate vulnerabilities. Feng et al. [8] intro-

duced Genius, a scalable feature extraction and vulnerability detection tool that uses static analysis to identify potential security issues by comparing firmware features to those of known vulnerable firmware.

3.2.2 Dynamic Analysis

Dynamic analysis techniques involve executing the firmware in a controlled environment to observe its behavior, enabling the detection of vulnerabilities that may not be apparent through static analysis alone. The primary dynamic analysis techniques used for firmware vulnerability detection include:

- **Emulation:** Emulation techniques create a software model of the hardware platform that the firmware is designed to run on, allowing the firmware to be executed in a safe environment where its behavior can be monitored for vulnerabilities. Koscher et al. [17] presented SURROGATES, a system for near-real-time emulation of embedded devices that enables communication with their peripherals through a custom FPGA bridge. Gustafson et al. [18] introduced PRETENDER, a system that automatically creates models of firmware peripherals to facilitate emulation by monitoring interactions with the original hardware. Kim et al. [19] developed FIRMAE, a tool capable of emulating a large dataset of firmware images, successfully emulating almost 80% of FIRMADYNE's dataset. Spensky et al. [20] created CONWARE, a framework that aids in the emulation of embedded systems by enabling the automatic generation of models for hardware peripherals. Zaddach et al. [21] presented Avatar, a dynamic analysis framework that emulates the peripherals of a physical device to facilitate firmware analysis. Chen et al. [11] proposed FIRMADYNE, the first system for automated dynamic analysis of Linux-based firmware, capable of extracting and emulating firmware filesystems to detect known and unknown vulnerabilities.

- **Fuzzing:** Fuzzing involves feeding the firmware with a large number of inputs to trigger vulnerabilities, to identify issues that may be difficult to detect with other

techniques. Zhu et al. [14] introduced FIoT, a framework for detecting memory corruption vulnerabilities through symbolic execution and fuzzing, which combines static and dynamic techniques to thoroughly test firmware security.

- **Rehosting:** Rehosting moves the firmware to a different hardware platform where it can be run in a more controlled environment, which is advantageous for large-scale dynamic analysis by avoiding the limitations of emulation techniques. Fasano et al. [22] provided a comprehensive overview and comparison of rehosting and emulation techniques, highlighting the benefits of rehosting for dynamic analysis of firmware on a large scale.

3.2.3 Symbolic Execution

Symbolic execution is a technique that explores the feasible execution paths of the firmware code, enabling the identification of vulnerabilities that may be difficult to detect with other methods, such as those that only occur under specific conditions. Subramanyan et al. [23] used symbolic execution for verifying firmware security properties, introducing a property specification language and a verification algorithm based on symbolic execution to ensure the confidentiality and integrity of firmware. Johnson et al. [24] presented Jetset, a system that deduces the behavior firmware expects from peripherals through symbolic execution, enabling the accurate emulation of firmware. Davidson et al. [25] developed FIE, a symbolic execution tool tailored for the MSP430 architecture, designed to find memory safety violations and peripheral misuse errors in firmware.

3.3 Evaluation & Comparison of Existing Tools

In this section, we compare several notable existing tools and frameworks for firmware vulnerability detection. We also highlight the advantages of our approach.

Redini et al. [4] created BootStomp, a tool that combines static analysis and dynamic symbolic execution to identify vulnerabilities in bootloaders. Although this tool is relevant

to our work, we encountered several challenges when attempting to integrate BootStomp into our analysis. Specifically, BootStomp was developed using outdated software versions: IDA Pro 6.95 and Python 2.7. Our attempts to update the tool for compatibility with newer versions of IDA Pro and Python were unsuccessful. Furthermore, we were unable to replicate the results reported by the authors. Despite our efforts to contact the authors via email and through an issue on their GitHub page [1], we did not receive a response. As a result, we concluded that BootStomp is out of date and not suitable for our needs.

Wen et al. [5] introduced FirmXRay, a tool for automatic static analysis of firmware to detect Bluetooth link layer vulnerabilities. While FirmXRay is relevant to our research, its scope is limited to BLE link layer vulnerabilities on TI and Nordic devices. Our proposed research has a broader scope.

Many previous works in firmware vulnerability detection (e.g., [6, 7, 8, 9, 1]) did not perform static information flow analysis. Instead, these works often relied on more basic static analyses such as string extraction, unpacking, and the collection of interesting files (e.g., authorized_keys files). Most of these approaches focus on identifying already known vulnerabilities by matching firmware code or versions with known vulnerable code (e.g., [8, 9]) or by determining if the firmware version is known to be vulnerable (e.g., [1]).

Table 3.1 provides an overview of the systems that perform vulnerability detection in firmware. The term "System" is used to refer to the implementation of the proposed techniques. We denote with an "N/A" followed by the first author's name and publication date, the systems that were not named by their authors. The systems are sorted in alphabetic order.

In Table 3.2 we present a comparison of the analysis techniques used by the systems that perform static analysis.

[1] https://github.com/ucsb-seclab/BootStomp/issues/12

Table 3.1: Overview of Existing Firmware Vulnerability Detection Systems

System Name	Supported Architecture	Sample Size	Analysis			Detected Vulnerabilities		
			Static	Dynamic	Symbolic Execution	Known	Unknown	Undefined
Avatar [21]	ARM	3		✓	✓		1	
BootStomp [4] *	ARM	5	✓		✓		8	
DTaint [12]	ARM, MIPS	6	✓		✓		21	
FIE [25] *	MSP430	99			✓			21
FIoT [14]	ARM, FreeRTOS	115	✓		✓		35	
FIRMADYNE [11] *	ARM, MIPS	9,486		✓		74	14	
Firmalice [26]	ARM, PPC	3	✓		✓		2	
FirmUp [9]	ARM, MIPS, PPC, Intel	∼2,000	✓			373		
FirmXRay [5] *	ARM	793	✓					2,146
Genius [8]	ARM, MIPS, Intel	8,126	✓			23		
Karonte [16] *	ARM	952	✓			5	46	
P^2IM [10] *	ARM	10		✓			7	
N/A (Costin'14) [6]	ARM, MIPS	32,000	✓				38	
N/A (Costin'16) [7]	ARM, MIPS	1,925	✓	✓			9,271	

* denotes that the system is open source

Table 3.2: Overview of Static Analysis Systems

System Name	Disassembler/ Implementation	Base Address Identification	Control Flow	Data Flow	Inter-Procedural	Constant Propagation	Multi-Binary	Implicit Flow
BootStomp [4]	IDA Pro, angr			✓	✓			
DTaint [12]	angr		✓	✓	✓			
FIoT [14]	IDA Pro, angr	✓	✓					
Firmalice [26]	Python, C	✓			✓			
FirmUp [9]	IDA Pro, angr		✓			✓		
FirmXRay [5]	Ghidra	✓	✓	✓	✓	✓		
Genius [8]	IDA Pro		✓					
Karonte [16]	angr		✓	✓	✓		✓	
N/A (Costin'14) [6]	BAT							
N/A (Costin'16) [7]	RIPS							

CHAPTER 4
METHODOLOGY

4.1 Overview

In this chapter, we detail the methodology employed by Xplicit, a system we designed to explore and identify potential vulnerabilities in ARM32 firmware. Our approach hinges on robust static analysis techniques, starting with the disassembly of firmware binaries and followed by conducting static information flow analysis. To enhance the depth and accuracy of our findings, we also incorporate implicit data flow analysis. This refined method allows Xplicit to meticulously trace how data propagates through the firmware, moving from input sources to hardware control registers, or sinks.

To systematically address the challenges of ARM32 firmware static information flow analysis, Xplicit initially focused on a targeted analysis of a thermostat's firmware, using it as a baseline to refine our techniques and workflow. This preliminary phase involved a detailed examination of two distinct firmware versions, allowing us to identify changes and potential vulnerabilities across these iterations. After establishing a robust analytical foundation with the thermostat firmware, we scaled our methodology to include a broader range of firmware samples and parallelized the analysis. This scaling was instrumental in testing the applicability and effectiveness of our analysis techniques across a wider array of devices and ARM32-based chipsets. By gradually expanding the scope of our analysis, Xplicit was able to optimize our tools and processes, ensuring they were capable of handling a larger number of firmware samples.

4.2 Infrastructure

4.2.1 Hardware

The backbone of our experimental setup is a lab server running Linux Ubuntu 20.04.3 LTS, chosen for its stability and compatibility with a wide range of analytical tools. This server is equipped with dual Intel Xeon E5-2450 CPUs, featuring a total of 16 physical cores and 32 threads, and 94GB of RAM.

4.2.2 Software Tools

Our analysis relies heavily on a suite of sophisticated software tools, each selected for its specific capabilities to support different aspects of firmware analysis:

- **IDA Pro v7.6** [27]: This disassembler is the cornerstone of our reverse engineering process, providing advanced capabilities to deconstruct ARM32 firmware into a more analyzable form. Its comprehensive support for ARM architectures and extensive feature set make it ideal for our analysis. Jiang et al. [28] performed an empirical study on the ARM disassembly tools existing. Based on their results IDA Pro has the best F1 Score and the best precision, while Hopper [29] has the best recall. This is why we chose to use IDA Pro as the most suitable disassembler for our analysis, ensuring both accuracy and reliability in our disassembly process.

- **Python & IDAPython**: We implemented our approach in the Python programming language. Python's standard library offers tools well-suited to many tasks and has been used many times to develop tools that analyze software [30, 31]. We used Python scripts extensively to automate repetitive aspects of the disassembly process and to script complex analyses. IDAPython is an IDA Pro plugin that allows us to run Python scripts in IDA Pro. This integration enhances IDA Pro's functionality, allowing us to customize and extend its capabilities to suit our specific research needs.

- **NetworkX** [32]: A Python package that we employed for creating and analyzing complex graphs of the firmware's control and information flows. NetworkX allows us to visualize data paths and interactions within the firmware. This tool is crucial for understanding how data moves through the system and for identifying potential security vulnerabilities.

- **GNU Parallel** [33]: To manage and optimize the processing of multiple firmware analyses simultaneously, GNU Parallel is utilized. This tool enables efficient use of our server's processing capabilities, significantly reducing the time required for comprehensive multi-firmware analysis.

4.3 Prototype Development

To develop our prototype, named *Xplicit*, our initial focus was on single-binary ARM32 firmware. We selected a thermostat manufactured by Beijing Brade Controls Tech Co., Ltd., as our DuT. This device operates on an ARM Cortex-M3 MCU, which is based on the 32-bit ARM v7-M architecture. The firmware is segmented into two binaries: the 'boot binary' and the 'main binary', distinct from the boot and steady states of the firmware.

We analyzed two versions of this firmware, v2.98 and v3.27, to track changes and identify potential vulnerabilities across updates. The tools configured for this analysis were outlined in the Infrastructure section; however, their application in the prototype development focused specifically on:

- **Custom Scripting and Automation with Python/IDAPython**: Enhancing the automation and repeatability of our analysis.

- **Accurate Memory Mapping with SVD Files**: Ensuring the physical configurations in our analyses align with the actual hardware.

- **Graph Creation and Analysis with NetworkX**: Creating the graphs based on the control and data flows of the firmware, and then analyzing them.

Our code, developed in IDAPython, enabled us to perform detailed Control Flow Analysis and Data Flow Analysis. The outputs were generated in SVG format to ensure the high resolution and clarity of the graph representations, crucial for detailed examination and presentation. Alternative output formats can be generated as needed, to support downstream analyses or integration with other tools.

4.4 Analysis at Scale

In this section, we detail the steps taken to scale up the automation and efficiency of our firmware analysis process, a key component of the Xplicit tool's functionality. Our scaling efforts focused on three core analytical techniques: Control Flow Analysis, Data Flow Analysis, and Data Flow Analysis with Implicit Flows.

To automate the disassembly and analysis of ARM binaries, we leveraged IDA Pro's ability to run in batch mode using command-line options. This approach allowed us to perform complex tasks like disassembly and code analysis without manually opening the program's graphical user interface, thereby streamlining the process and reducing manual intervention. By doing so, we successfully automated both the Control Flow and Data Flow processes, resulting in a scalable system capable of processing each firmware binary within seconds for the Control Flow and minutes for the Data Flow Analysis.

The enhancements were not just limited to automation. We also developed the capability to process multiple firmware samples in parallel within the Xplicit tool by using GNU Parallel. This significant upgrade streamlined our analysis procedures, enabling the simultaneous processing of several samples and dramatically improving our research throughput.

These advancements have substantially increased the effectiveness and efficiency of the Xplicit tool, establishing it as a powerful solution for comprehensive firmware evaluation at scale. The ability to concurrently process multiple firmware samples can greatly enhance our capacity to identify potential vulnerabilities across a wide spectrum of devices.

Currently, this process works for ARM ELF binaries. Since we are building a prototype

of Xplicit, we did not want to add extra complexity by loading binaries for which we do not have any information. Besides, the problem of properly loading a binary file has already been studied and solved to a satisfying degree in previous work [5].

4.5 Control Flow Analysis

In this section, we detail the methodology employed to develop the Control Flow Graph (CFG) using IDAPython, an essential component of our analysis toolkit. The development of the CFG required writing custom code that interacts extensively with IDA Pro's disassembly capabilities. Our IDAPython script initiates by iterating over all the instructions within each function as identified by IDA Pro. For each instruction, the script determines the function to which it belongs. It then systematically identifies all cross-references from the current function to other instructions, enumerating these instructions in the order they are called. Importantly, the script only adds instructions to the CFG if they are part of an identified function. If an instruction that is cross-referenced does not belong to any function, it is excluded from the graph. Similarly, if our analysis encounters an instruction outside of any function context, it does not proceed to find its cross-references.

One notable challenge in this process is IDA Pro's occasional inability to correctly assign code to functions. To address this for the thermostat firmware and enhance the accuracy of function determination, we used the linear sweep disassembling algorithm [34]. This algorithm aids in correctly placing orphaned code into appropriate functions, thereby improving the comprehensiveness and reliability of our Control Flow Analysis.

A primary use of the CFG is to assist in distinguishing between the boot state and the steady state of the firmware. To the best of our knowledge, there is no defined separation between these states in the literature. Therefore, we have established a process based on specific assumptions about the firmware behavior. We assume that after the initialization of the Control Registers and Peripherals during the boot state, the firmware enters a loop of steady operation without exit options, which we consider as the steady state of the firmware.

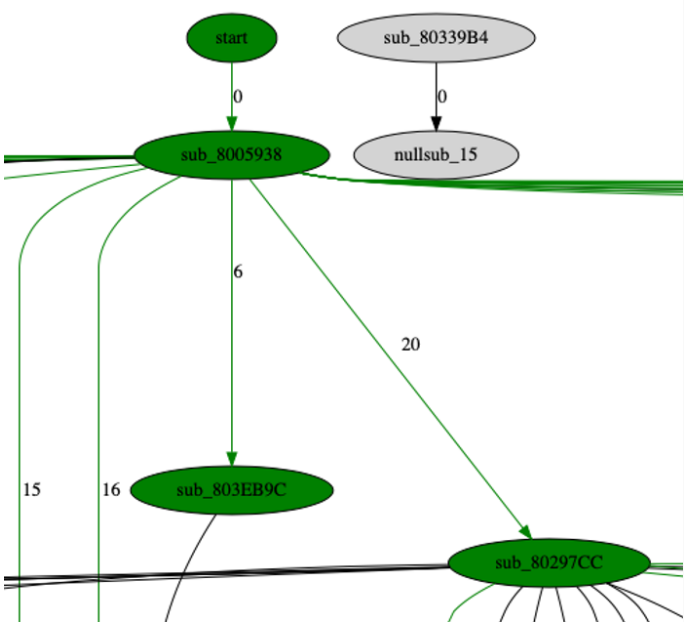

Figure 4.1: Part of the Control Flow Graph created with Xplicit for v3.27 of the DuT

Accordingly, we define the steady state as the first reachable non-returnable function from the start of the boot process. This function is expected to be depicted in the CFG as a root that branches out into a tree covering the steady-state operations. Furthermore, the CFG can be utilized to identify the interrupt handlers within the firmware, providing a deeper understanding of its structure and operation.

4.5.1 Control Flow Graph Visualization

We visualize the control flow analysis through a graph model, as depicted in Figure Figure 4.1. The nodes in the graph represent functions within the firmware, and the edges illustrate the control flows between these functions. Each node is labeled with the function name or identifier assigned by IDA Pro, and each edge is labeled to show the sequential order in which functions are called. The direction of each edge indicates the flow of control.

The visual representation of the CFG employs a color-coded system designed to enhance the clarity and readability of the analysis. This system, which we developed, uses distinct colors to indicate various functional aspects of the firmware, aiding in quick visual assessment and understanding:

- **Green nodes**: Represent functions that are part of the execution path starting from the main function, highlighting active paths in the firmware's operation.
- **Grey nodes**: Denote functions called before the firmware reaches the main function, illustrating the initialization phase of the firmware.
- **Red nodes**: Highlight functions that are unreachable from the main function's execution path, typically representing interrupt handlers or unused code segments.
- **Green edges**: Indicate feasible paths originating from the main function, showing potential flows of execution within the firmware.
- **Black edges**: Show paths that are not reachable from the main function, which may indicate conditional or exceptional flows that are not activated in normal operation.

This approach to visualizing the control flow simplifies the identification of key operational features in the firmware's execution.

4.6 Data Flow Analysis

In this section, we discuss the Data Flow Analysis techniques implemented in Xplicit, which play a pivotal role in mapping the propagation of data within firmware binaries. Our approach leverages Use-Definition (Use-Def) and Definition-Use (Def-Use) chains, which are fundamental in understanding instruction dependencies and generating the Data Flow Graph (DFG). Def-Use chains link definitions of variables to their points of use, whereas Use-Def chains trace the definitions of variables used in instructions.

Our scripts are robust, designed to function with or without specific chipset information:

- If the specific chipset is known and an SVD file is available, we import it into IDA Pro. This import process helps accurately rename the firmware segments, aligning them with the physical hardware layout.

- If only the family of the processor is known (e.g., Cortex-M3), without specific chipset details, we manually configure the processor's memory mapping in IDA Pro to simulate the appropriate segments.

To construct the DFG, our custom IDAPython scripts analyze the firmware as follows, depending on the availability of the SVD file:

1. **With SVD**:

    - The scripts traverse memory addresses reserved for peripherals and hardware control registers (Addresses $\geq$ 0x40000000) as categorized by IDA Pro after processing the SVD file.
    - We specifically focus on instructions that are labeled during the SVD application by IDA Pro, as these typically interact with actual peripheral and hardware control registers.
    - Cross-references to these labeled instructions are examined, and Data Flow Analysis is performed within their respective functions.

2. **Without SVD**:

    - Instructions within the code segment, which typically contains the core code of the firmware, are iterated (Addresses $\leq$ 0x20000000).
    - For instructions utilizing 'LDR' commands, we check if they have a cross-reference to an address of a hardware control register or peripheral.
    - If an instruction satisfies this check, we perform Data Flow Analysis for the function to which this instruction belongs.

To enhance our understanding of the expected memory segments and gain insight into the memory mapping of the chipsets used in the Devices Under Test (DuTs), we provide a consolidated memory mapping for Cortex-M3 and Cortex-M4 processors, both based on

the ARM v7-M architecture. All the DuTs analyzed in our study are equipped with either Cortex-M3 or Cortex-M4 chipsets. The memory mapping, detailed in Table 4.1, outlines the address ranges for code segments, RAM, peripherals, and other essential system areas, which are pivotal for our data flow analysis.

Table 4.1: Memory Mapping for Cortex-M3 and Cortex-M4 ARM Processors

Segment	**Address Range**
CODE	`0x00000000 - 0x1FFFFFFF`
RAM	`0x20000000 - 0x3FFFFFFF`
PERIPHERAL	`0x40000000 - 0x5FFFFFFF`
EXTERNAL DEVICE	`0xA0000000 - 0xDFFFFFFF`
PRIVATE PERIPHERAL BUS (Internal)	`0xE0000000 - 0xE003FFFF`
PRIVATE PERIPHERAL BUS (External)	`0xE0040000 - 0xE00FFFFF`
SYSTEM	`0xE0100000 - 0xFFFFFFFF`

4.6.1 Data Flow Graph Visualization

We visualize the Data Flow Analysis through a graph model, as depicted in Figure Figure 4.2. Unlike the Control Flow Graph (CFG) which operates at the function level, the Data Flow Graph (DFG) operates at the instruction level, leading to distinct visual representations.

In the DFG, nodes represent individual instructions, and edges denote the data flows between them. Each node is labeled with the corresponding instruction address followed by the instruction itself. Edges are labeled with the variable (e.g., general-purpose register) through which the data flows from one instruction to another, with arrows indicating the direction of the flow.

Additional visual elements in the DFG provide insights into the data flow characteristics:

- **Green nodes**: Represent LDR instructions that load the address of a peripheral or hardware control register into general-purpose registers. They mark the initial access to a peripheral's address.

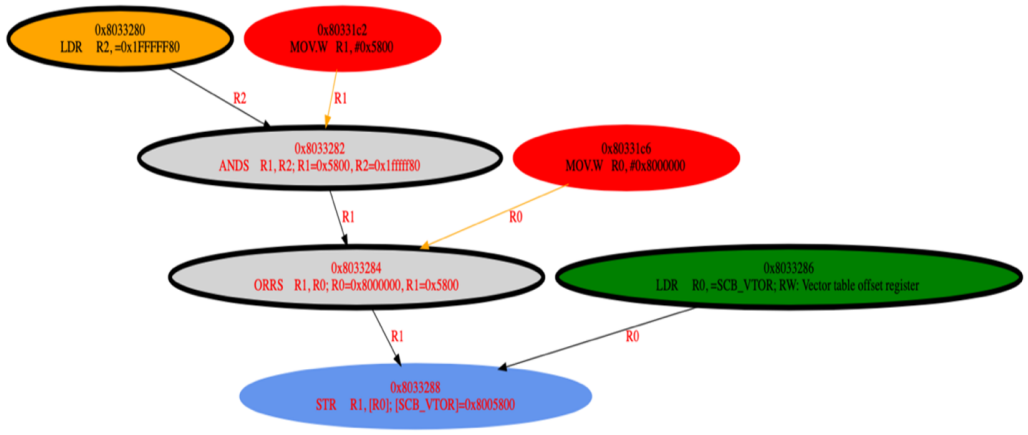

Figure 4.2: Example of a Data Flow Graph created with Xplicit for firmware version v3.27 of the (DuT)

- **Light blue nodes**: Indicate instructions where there is direct access or use of a peripheral or hardware control register.
- **Red nodes**: Denote source instructions where immediate values are introduced into the data flow, influencing subsequent operations.
- **Orange nodes**: Similar to red nodes, but specifically involve 'LDR' instructions, typically indicating the reading from a peripheral or hardware control register.
- **Gray nodes**: Intermediate nodes that alter the data values or registers being tracked.
- **Orange edges**: Represent data flows originating from different functions, highlighting inter-procedural interactions. Inter-procedural analysis was a feature that we added later on in the data flow analysis, but we still thought it would be better to mention it here. More details are given in the next section, section 4.7.
- **Blue edges**: Indicate the existence of an implicit data flow. These edges do not have labels, as the flow of data is implicit and not explicit, unlike the other edges which show explicit data flows. Like the inter-procedural analysis, implicit flows were a feature that we added later on in the data flow analysis, but we still thought it would be better to mention it here. More details are given in section 4.8.

Figure Figure 4.2 showcases a subgraph from our DFG, illustrating each visual element employed in our analysis. This example underscores the flow of data through specific instructions, providing a clear depiction of how variables and registers are manipulated across different operations within the firmware. Specifically, we observe the following:

- **0x8033286**: The hardware control register SCB_VTOR (Vector Table Offset Register) is loaded into the general-purpose register R0 in instruction 0x8033286, therefore, this node is colored green to denote this operation.
- **0x8033288**: SCB_VTOR is accessed in instruction 0x8033288, indicated by a light blue node which signifies access of a control register.
- **0x8033284**: R1, used in instruction 0x8033288, is defined at instruction 0x8033284. The value in R0, coming from another function "sub_80331C0" (visualized with an orange edge), originates from instruction 0x80331c6, which assigns an immediate value to R0, thus the node is red. R1 originates from instruction 0x8033282.
- **0x8033282**: The definition of R1, used in instruction 0x8033282, occurs in instruction 0x80331c2. This instruction assigns an immediate value to R1, therefore, the node is painted red. This flow of information originates from another function, "sub_80331C0," and is represented with an orange edge. Additionally, R2 used in instruction 0x8033282 is defined at instruction 0x8033280, which assigns the value 0x1FFFFF80 to R2. This value could be either an immediate value or the address of a hardware control register, hence, it is colored orange.

4.7 Enhanced Data Flow Analysis

After developing the first version of our Xplicit prototype, we performed multiple iterations to improve its data flow analysis. These improvements included the incorporation of interprocedural analysis, constant propagation, control register and peripheral discovery, identifying access patterns of the control registers and peripherals, detection of control registers

and peripherals without SVD, and implicit data flow analysis. By leveraging these techniques, our methodology provides a deeper and more comprehensive analysis of firmware binaries, ensuring a more robust identification of potential vulnerabilities.

4.7.1 Inter-procedural Analysis

Initially, our approach could only track data flow within individual functions, which limited the scope of our analysis. By incorporating inter-procedural analysis, we significantly increased the size and complexity of the DFGs, allowing us to track information flow across multiple functions. Figure Figure 4.2 illustrates these inter-procedural flows with orange-colored edges, highlighting the data flow from instructions 0x80331c2 and 0x80331c6 in function "sub_80331C0" to instructions 0x8033282 and 0x8033284 in function "sub_8033280".

4.7.2 Constant Propagation

To further enhance our data flow analysis, we implemented basic constant propagation. This technique helps determine whether the value flowing to a control register or peripheral originates from an immediate value. This distinction is crucial because if data originates from an immediate value, the behavior is fixed. In contrast, if data does not come from an immediate value, the source is potentially a peripheral, which is a primary focus of our analysis. Thus, an analyst looking at the DFG can quickly ignore data that comes from an immediate value.

We handle the most common classes of instructions found in most cases. Specifically, we handle the "MOV", "LDR", "ORR", "AND", and "LSL" instructions. Our code is modular, allowing for the addition of more instructions by implementing the corresponding functionality.

Constant propagation enhances our analysis in cases where constants propagate to a control register or peripheral instead of a general-purpose register. Figure Figure 4.3 depicts

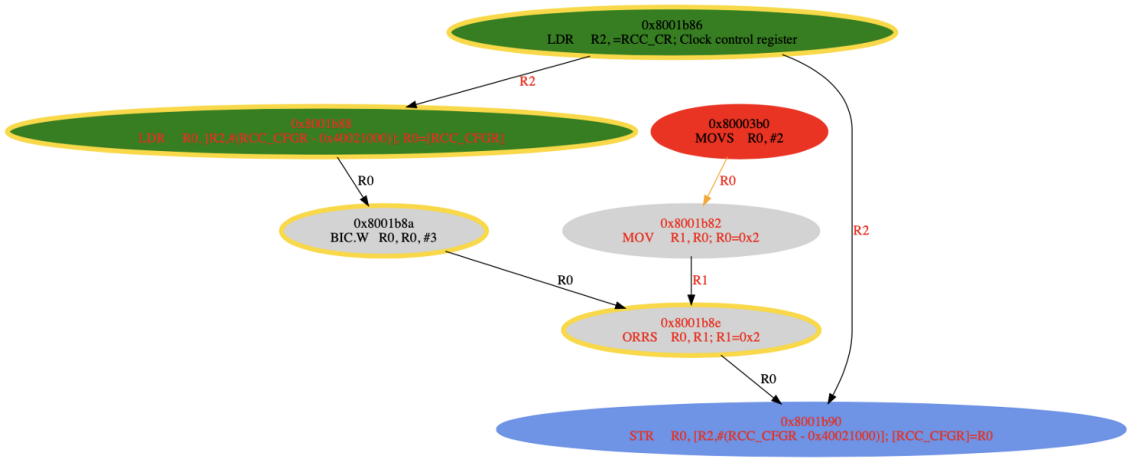

Figure 4.3: Constant Propagation with Xplicit for v3.27 of the DuT

a subgraph of the DFG containing constant propagation analysis. In instruction 0x8001b86, the control register named RCC_CR (clock control register) is loaded into the general-purpose register R2. Then, in instruction 0x8001b90, the value of the general-purpose register R0 is stored into [R2,#(RCC_CFGR - 0x40021000)].

Through constant propagation, we know that R2 is RCC_CR, corresponding to the memory address 0x40021000. RCC_CFGR (clock configuration register) corresponds to the memory address 0x40021004. Thus, the calculation performed is:
$0x40021000 + 0x40021004 - 0x40021000 = 0x40021004$. RCC_CFGR is located at 0x40021004, indicating that this instruction stores the value of the general-purpose register R0 into RCC_CFGR.

4.7.3 Control Register & Peripheral Discovery

Initially, our code could only track uses (”STR”) that occurred immediately after a control register or peripheral was loaded (”LDR”) into a general-purpose register. Manual inspection of the output graphs revealed that the data flow was often incomplete, failing to accurately represent the binary’s data flow. The root cause was that if a control register or peripheral was loaded into a general-purpose register and its value changed subsequently, our code did not track this information flow, leading to its exclusion from the DFG.

To address this issue, we enhanced our methodology to track changes in the values of general-purpose registers after they have been loaded with control registers or peripherals. This enhancement significantly improved our understanding of how data flows within the firmware of the DuT, as we show later in the results.

4.7.4 Identification of Control Registers & Peripherals without SVD

One capability that we added to our tool is the ability to perform the analysis without relying on SVD file data, as described in Data Flow Analysis. We enhanced our methodology to identify control registers and peripherals even in the absence of SVD files.

Our approach allows us to detect more control registers and peripherals without the SVD file compared to when the SVD file is applied. This is because some peripherals are not recognized by the SVD file, resulting in IDA Pro not generating the corresponding segments in its memory mapping. By iterating over these addresses directly, we ensure that such control registers and peripherals are not overlooked.

4.8 Implicit Data Flow Analysis

We introduced the concept of implicit flow analysis with an example in Listing 2.1. After incorporating implicit flows into our analysis, we observed that they add negligible overhead to the analysis time of our tool while providing more comprehensive information. An illustration of our implicit data flow analysis can be seen in Figure Figure 4.4.

In our visual representation, a blue edge indicates the existence of an implicit data flow. The blue edge does not have a label, as the flow of data is implicit and not explicit, unlike the other edges which show explicit data flows.

As we show later in the results, the introduction of implicit flows into our analysis led to significant enhancements.

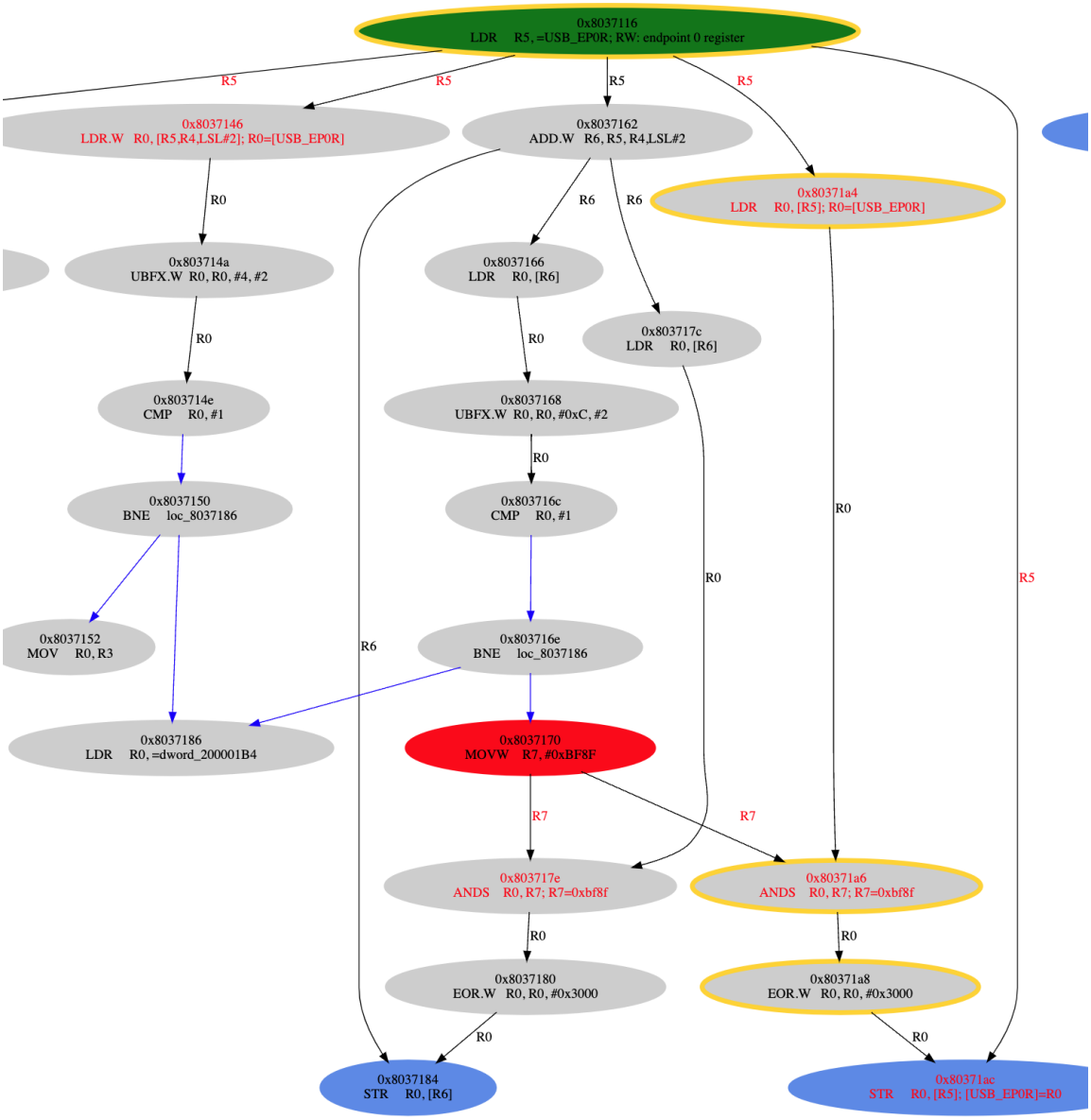

Figure 4.4: Implicit Data Flow Analysis created with Xplicit for v3.27 of the DuT

CHAPTER 5

RESULTS

5.1 Prototype Development Results

5.1.1 Separation of Boot & Steady State

In Figure 4.1, we observe that the function "sub_8005938" is called from the start function. This function serves as the root of a large tree that branches out into multiple functions, indicating that all these functions are called from "sub_8005938". This behavior aligns with our expectation for identifying the beginning of the steady state, suggesting that the steady state begins within the function "sub_8005938".

To locate the exact point where the steady state starts, we examine IDA Pro's CFG depiction of this specific function. IDA Pro's CFG provides details at the basic block level, allowing us to pinpoint the details of the function "sub_8005938".

In Figure 5.1 we see that function "sub_8005938" branches out and links initially to the following functions "sub_803ECB4", "sub_80269B8", "sub_8027960", "sub_80088B8", "sub_8006950", "sub_8027A74", "sub_803EB9C" and "sub_80373FC". Afterward, the control flow passes to "loc_803D526". We can also observe that the firmware enters a loop when it reaches "loc_803D526". Given this control flow pattern, we determine that the steady state begins at "loc_803D526". All control flows preceding "loc_803D526" are considered part of the boot state.

5.1.2 Control Register & Peripheral Discovery Results

The improvement in tracking control registers and peripherals led to a substantial increase in the size and completeness of our data flow graphs. After implementing this change, our graphs grew by more than 100%, as shown in Table Table 5.1. In the boot binary, the

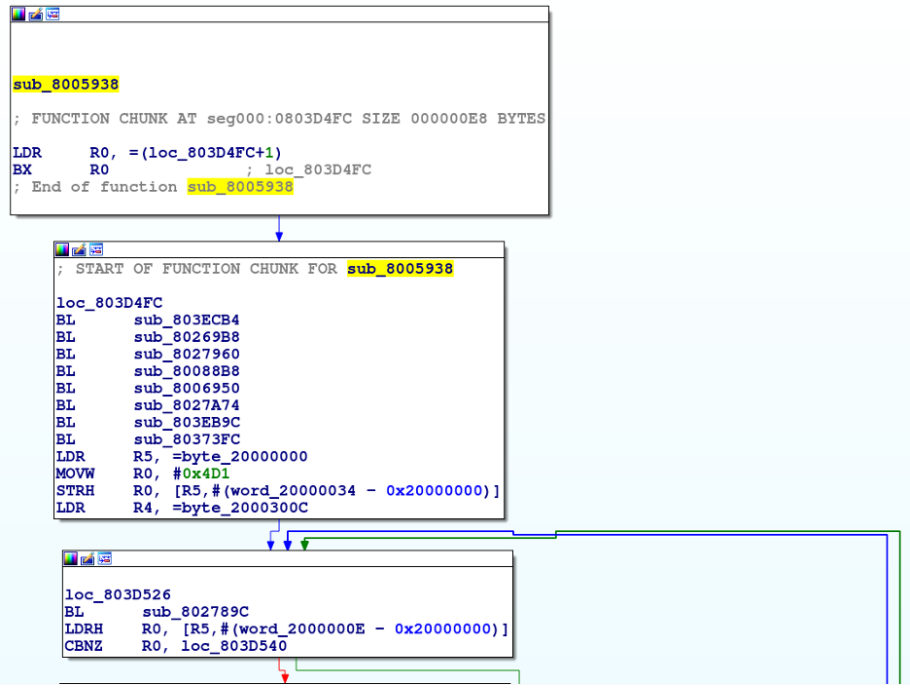

Figure 5.1: Control Flow Graph from IDA Pro for function sub_8005938 of DuT's v3.27

number of nodes increased from 466 to 974, and edges from 422 to 867. Similarly, in the main binary, nodes increased from 354 to 762, and edges from 414 to 883.

Table 5.1: Improvement in the Discovery of Control Registers & Peripherals

Binary	#	Before	After	Increase
Boot	Nodes	466	974	109.01%
	Edges	422	867	105.45%
Main	Nodes	354	762	115.25%
	Edges	414	883	113.29%

5.1.3 Identification of Control Registers & Peripherals without SVD Results

The ability to detect control registers and peripherals without the SVD file significantly improved the comprehensiveness of our analysis. We observed that our tool identified more control registers and peripherals when the SVD file was not applied.

As we show later in the results, Appendix A provides the analytical validation of these findings, demonstrating the increased detection capabilities and highlighting specific in-

stances where peripherals were not recognized by the SVD file but were successfully identified through our enhanced methodology.

5.1.4 Defining Performance of Identifying Control Registers & Peripherals

Our refined approach has enabled the precise identification and analysis of all the control registers and peripherals used by the firmware of the DuT.

Utilizing the reference manual for the Cortex-M3 and the STM32F103ZE chipset, along with the SVD for this chipset, was crucial.

Furthermore, we employed the memory mapping of the Cortex-M3 to conduct a broader analysis of the firmware, even without specific knowledge of the chipset used.

Performance analysis is achieved using a metric that provides a rate of success as a factor of control registers identified. This metric value provides the success rate in terms of percent value (%).

$$\frac{CR\_DFG}{CR\_IDAPro} * 100\%, \tag{5.1}$$

Equation 5.1 defines "CR_DFG" as the control registers found through the DFG, and "CR_IDAPro" represents the control registers identified via IDA Pro. Our results show a 100% success rate in identifying the control registers regardless of the knowledge of the specific chipset used by the firmware. The high success rate bolsters confidence in the research approach and thoroughness of the analysis.

5.1.5 Implicit Data Flow Analysis Results

The introduction of implicit data flows into our analysis led to significant enhancements. Table Table 5.2 compares the graphs before and after the inclusion of implicit data flows.

We observe that the number of nodes and edges increased by more than 4% and 6%, respectively. The number of connected components decreased, while the size of the largest component increased. Although this might seem counter-intuitive, it indicates that some

existing components merged to form larger components, leading to more complete information about the data flow in the firmware.

Table 5.2: Before and after the Discovery of Implicit Data Flows

Binary	Attribute/Feature	Before	After	Change
Boot	Nodes	975	1021	4.72%
	Edges	868	927	6.80%
	Connected Components	155	150	-3.23%
	Largest Connected Component Size	40	57	42.50%
	Analysis Time (s)	115.70	116.77	0.92%
Main	Nodes	807	842	4.34%
	Edges	975	1035	6.15%
	Connected Components	82	81	-1.22%
	Largest Connected Component Size	84	109	29.76%
	Analysis Time (s)	794.29	794.63	0.04%

5.2 Identifying Risky Flow in Firmware

5.2.1 Threat Model

To prove the value of our tool, we propose the following threat model.

We assume an attacker that has gained access to the controller device and tries to alter the state of the worker device by sending requests to it through the Modbus communication protocol. The attacker can send an arbitrary number of messages (function codes) to the DuT and make it operate in an unintended way. Examples of such attacks have been described in [35]. A message sent through the Modbus protocol usually contains the Protocol Data Unit (PDU) function code, a register address, a value to be written to that address, as well as the worker's address and an error check (checksum). All the elements above constitute the Application Data Unit of the Modbus protocol [36]. A successful attack must meet two conditions to be successful:

1. the Modbus message must write a value to a specific hardware control register of the firmware.

2. this value can change the state of the DuT when written to that register.

Finally, we assume that attackers know the internals of the DuT chipset so that they can select the hardware control registers they want to target.

5.2.2 Risky Flow

We have already defined information flow as the propagation of information from a source to a sink. We now define risky flow as an information flow that has an external input as a source (e.g., data coming from a protocol the DuT uses for its communication) and reaches a sink that is a hardware control register that can change the state of the DuT in an unintended way. We used our tool to help us analyze the firmware of an IoT device and identify a potentially risky flow that could compromise the security of the DuT based on the threat model mentioned in subsection 5.2.1. For our analysis we used the older firmware version of the DuT (v2.98).

In function sub_800D5BC (shown in Figure 5.2), we can see an example of how our tool helps to identify data/signal in control register UART5_SR (source) that depends on data coming from the Modbus protocol that impacts the hardware control register USB_EP0R (sink). Specifically, we see that only if *Condition_1* in instruction "CBZ R3, loc_800D65A" is not satisfied, the control flow will pass to *Block_1* (loc_800D678), which updates the value of USB_EP0R. We consider this flow risky based on our definition since UART and USB are two peripherals that should operate independently. Therefore, such a flow could cause unpredictable behavior and potentially disrupt the USB functionality.

There are more conditions that need to be satisfied so that the control flow will pass to *Block_1*, but they are dependent on data coming from other sources (e.g., data in RAM). Since our analysis is only static, we assume that the rest of the conditions are met and the described path is feasible. To validate our assumptions, dynamic analysis of the firmware would be needed.

In general, a risky flow can arise from an explicit information flow. However, with our

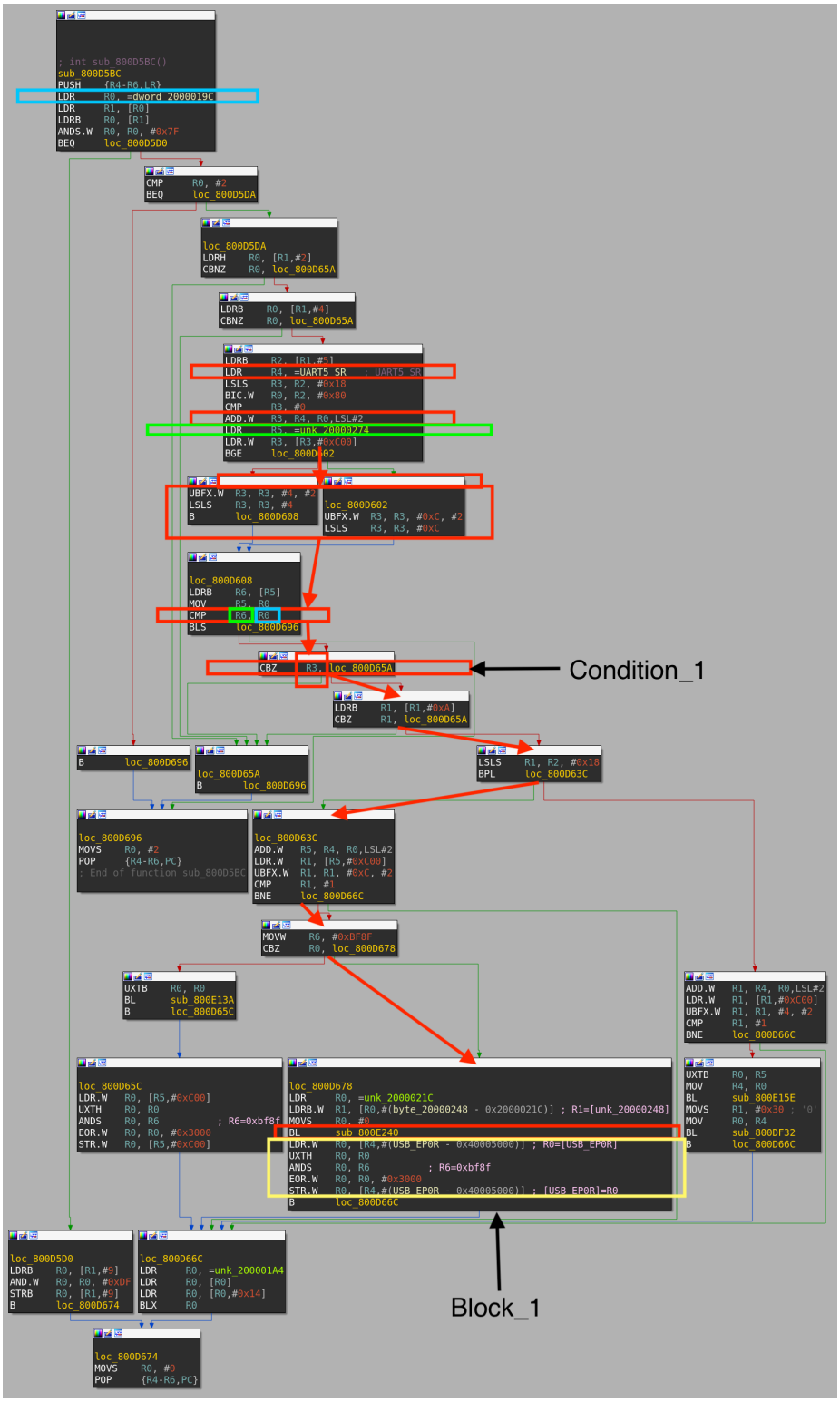

Figure 5.2: View from IDA Pro for function sub_800D5BC of DuT's v2.98

findings, we show that there can be a risky flow arising from implicit information flow. Previous works do not identify such risky flows making our research the first to identify them.

Leveraging the threat model and the risky flow we identified, we propose an attack scenario that could compromise the DuT's security. We assume that we have the IoT thermostat, and it is a worker device in an ICS. The controller device can send messages to the worker through the Modbus protocol. An attacker would first try to gain unauthorized access to the controller device. After gaining access and identifying the DuT in the network, they can attempt to exploit vulnerabilities in its firmware. Based on our findings, this can be done by sending data to the UART peripheral, to write into the USB_EP0R hardware control register after propagation, and disrupting the USB functionality. This could have one of the following outcomes:

1. The disruption causes a buffer overflow or memory leak, an attacker could use this vulnerability now to execute malicious code on the device or gain unauthorized access to sensitive data.

2. The disruption could potentially prevent firmware updates or configuration changes from being made via USB, or prevent the thermostat from reporting data to a USB-connected monitoring system.

3. The attacker uses the disruption as a diversionary tactic to distract security personnel while they attempt to exploit another vulnerability in the system.

Based on our threat model, this could be a successful attack since (1) the Modbus message could write a value to a specific hardware control register of the firmware, and (2) this value could change the state of the DuT when written to that register (by disrupting the USB functionality).

5.3 Analysis at Scale Results

In this section, we present the results of analyzing multiple firmware samples in parallel, a key capability of the Xplicit tool. We were able to successfully analyze 8 firmware samples concurrently. The details of the graphs produced by Xplicit are summarized in Table 5.3.

Our generalized approach effectively managed a diverse array of firmware samples from various devices including a Door Lock, a Drone, a Robot, a CNC Router, a PLC, a Steering Control, and the MC6 thermostat used in the prototype development phase. These firmware binaries were sourced from a GitHub repository [37] maintained by UCSB SecLab and include many that were originally analyzed in the P2IM study [10]. This selection demonstrates the robustness of our methods across a range of applications and chipsets, initially including and building upon our earlier analyses of two thermostat firmware binaries. All the chipsets were of the 32-bit ARM v7-M architecture and used Cortex-M3 or Cortex-M4 processors. Most of the chipsets were manufactured by STMicroelectronics, but we also had a chipset from Microchip Technology Inc. (previously known as Atmel).

The analysis of each firmware sample was conducted using the Control Flow Graph (CFG), Data Flow Graph (DFG), and Data Flow Graph with Implicit Data Flows (DFG with Implicit Data Flows) techniques. The results include metrics such as the number of nodes, edges, and connected components for each type of graph, as well as the analysis time in seconds. For DFG with Implicit Data Flows, the implicit paths discovered are also included as a metric. These metrics provide a view of the complexity and scalability of our analysis approach.

Challenges persisted in achieving complete automation of the Data Flow Analysis. Specifically, the automation was fully achievable only in scenarios where importing an SVD file was not necessary. When SVD files were required, the limitations of IDA Pro's command-line interface necessitated manual intervention to import these files through the GUI, which hindered full automation. However, for binaries that did not require SVD files,

our Data Flow Analysis was entirely automated.

Overall, the results demonstrate the robustness and efficiency of Xplicit in handling a variety of firmware samples, showcasing its potential for large-scale firmware analysis.

Table 5.3: Xplicit Analysis at Scale Results

DuT	Processor	Chipset	Binary Size	CFG				DFG				DFG with Implicit Data Flows				
				Analysis Time (s)	Nodes	Edges	Connected Components	Analysis Time (s)	Nodes	Edges	Connected Components	Analysis Time (s)	Nodes	Edges	Connected Components	Implicit Paths
MC6 v2.98	Cortex-M3	STM32F103ZE	1.4M	45	978	3282	12	769	1878	1966	238	778	1956	2074	233	40
MC6 v3.27	Cortex-M3	STM32F103ZE	1.1M	34	1018	3463	8	873	1782	1843	237	881	1863	1962	231	21
Door Lock	Cortex-M3	STM32L152XE	331K	4	304	499	3	510	797	954	36	495	830	1005	32	20
Drone	Cortex-M3	STM32F103RB	412K	12	212	394	2	158	479	583	27	145	508	622	26	16
Robot	Cortex-M3	STM32F103C8	962K	2	189	297	1	172	661	705	68	128	738	806	56	35
CNC Router	Cortex-M4	STM32F429ZI	299K	4	290	571	13	173	1208	1404	95	173	1335	1578	78	60
PLC	Cortex-M4	STM32F429ZI	763K	1	280	356	8	807	888	1217	47	795	975	1341	30	64
Steering Control	Cortex-M3	ATSAM3X8E	271K	1	190	248	5	29	168	194	21	26	174	206	21	9

CHAPTER 6
DISCUSSION

In this chapter, we discuss the limitations of our current approach and propose future work to enhance the capabilities of our firmware analysis tool, Xplicit. Understanding these limitations is crucial for setting realistic expectations and identifying areas for improvement. Additionally, outlining future work helps in providing a roadmap for subsequent research and development.

6.1 Limitations

6.1.1 Control Flow Graph Limitations

The production of the CFG using our tool, Xplicit, encounters certain limitations that impact its accuracy and completeness. These limitations are inherent to the process of automated disassembly and analysis, and include:

1. **Code Recognition**: Code recognition is a well-known challenge across disassemblers, including IDA Pro. This impacts our ability to accurately identify and process all executable code within the firmware.

2. **Function Recognition**: If IDA Pro does not recognize certain segments as code belonging to a function, and these segments are not manually assigned to a function, then our analysis does not include these segments. This results in potential gaps in the CFG.

3. **Reachability of Functions**: There are functions in the firmware that are not reachable. Currently, we consider these functions to be interrupt handlers. Another possibility for not reaching these functions can be attributed to IDA Pro's automated

analysis of the firmware, where some code might not be recognized properly as code.

4. **Call Order Accuracy**: The call order depicted in the CFG might not always reflect the actual execution path, especially when branching is conditional. Moreover, functions called multiple times within another function are only recorded at their first invocation, omitting subsequent calls from the visualization.

5. **Points-to-Analysis**: Our current methodology does not use points-to-analysis, which could predict certain jumps or calls by understanding where general-purpose registers are pointing. This omission can lead to incomplete flow paths in the CFG.

These limitations delineate the scope of our CFG's reliability and should be considered when interpreting its output. Enhancements to address these challenges are potential areas for future work.

6.1.2 Data Flow Graph Limitations

A primary limitation in our DFG analysis arises from the ambiguity in distinguishing between immediate values and hardware control register addresses when using 'LDR' instructions. This uncertainty can lead to inaccuracies in the DFG, where nodes might incorrectly represent the loading of control registers or peripheral addresses instead of immediate values. This could potentially lead to misinterpretations of the data flows within the firmware.

Additionally, there are control registers and peripherals that are not identified through our analysis, attributed to control registers directly assigned into general-purpose registers with a "MOV" instruction (e.g., `MOV.W R7, #0xE000E000`). Our tool is only searching for control registers and peripherals that are initially loaded with "LDR" instructions to have a more focused analysis.

6.1.3 Implicit Data Flow Analysis Limitations

Our prototype creates a path representing an implicit flow by connecting the conditional instruction to the immediately next instruction where the control flow would branch after the execution of the comparison check. However, upon review, we noticed instances where the comparison check was followed by a block of code rather than a single instruction. In such cases, it would be more reasonable to connect the conditional instruction to each instruction in the subsequent block to avoid losing information from our data flow analysis.

The problem with introducing too many implicit flows, though, is that it would create many new paths, most of which add no value to our targeted analysis; hence, the whole effort for a more targeted analysis would be defied.

6.1.4 Static Information Flow Analysis Limitations

Always when performing static analysis, one has to keep in mind that dynamic analysis is needed to verify that a potential vulnerability is an actual vulnerability. Static analysis can identify paths that might lead to vulnerabilities, but to make sure that the path leading to the vulnerability is feasible, actual execution of the firmware code is needed. This limitation highlights the necessity of dynamic analysis to confirm the practical relevance and accuracy of the identified potential vulnerabilities.

6.2 Future Work

6.2.1 Decluttering Data Flow Graphs

We let for future work to declutter the produced DFGs. Currently, manual inspection of hundreds of nodes is required to identify potential vulnerabilities. To make our system even more scalable, we need to analyze more firmware samples, to manually identify the paths of potential vulnerabilities, and then determine the access patterns of the hardware control registers that indicate potential vulnerabilities. A good starting point to scale the analysis

is the [37], containing more than 800 firmware binaries. Subsequently, there should be removal of the remaining paths that do not follow these identified access patterns from our DFGs.

6.2.2 Enhanced Implicit Data Flow Analysis

To address the implicit data flow limitation, an enhanced implicit flow analysis could be implemented using heuristics to select and fetch instructions that add potentially ”risky” paths to the analysis, indicating possible risks in the firmware. Currently, our prototype connects the conditional instruction to the immediately next instruction, which may result in the loss of valuable information when the comparison check is followed by a block of code rather than a single instruction. Analyzing more firmware samples could help in determining appropriate heuristics, refining the approach, and ensuring a more comprehensive understanding of implicit data flows.

6.2.3 Usability Testing and User Study

We believe our tool is more usable than existing tools and enables even less technical individuals to perform firmware analysis. However, a user study is needed to test the usability of our tool and validate this claim. Conducting such a study will provide valuable insights into how users interact with Xplicit and identify areas where the user experience can be improved.

CHAPTER 7
CONCLUSION

In this thesis, we focus on firmware analysis aimed at detecting potential vulnerabilities in ARM32 firmware. To address current methodological gaps, we introduce Xplicit, a tool designed to enhance the effectiveness of firmware analysis by integrating both explicit and implicit data flow analysis. Xplicit is built on top of IDA Pro and IDAPython for disassembly and scripting, NetworkX for control and data flow visualization, and GNU Parallel for scalability. Through the novel development of a static information flow analysis method that identifies implicit data flows within ARM32 firmware, Xplicit offers a robust and scalable approach. Last but not least, Xplicit reduces its reliance on IDA Pro after the initial disassembly, making control and data flow analysis more accessible to users with varying technical backgrounds.

Appendices

APPENDIX A

CONTROL REGISTERS & PERIPHERALS IDENTIFICATION

In Table A.1, we observe the address of the peripherals, their type (based on the Cortex-M3 system address map), the name based on the SVD file, and then the binary values boot and steady, which represent the state of the firmware. If the value is 1, that means that we have detected this control register or peripheral in the corresponding state, otherwise, the value is 0. If the name is "NOT IN SVD" that means that this control register or peripheral was only detected by our analysis without the SVD.

Table A.1: Control Registers & Peripherals Identified

Address	Type	Name	Boot	Steady
0x40003000	Peripheral	IWDG_KR	1	1
0x40004800	Peripheral	NOT IN SVD	0	1
0x40004804	Peripheral	NOT IN SVD	0	1
0x40005c00	Peripheral	USB_EP0R	0	1
0x40005c40	Peripheral	USB_CNTR	0	1
0x40005c44	Peripheral	USB_ISTR	0	1
0x40005c50	Peripheral	USB_BTABLE	0	1
0x40007400	Peripheral	DAC_CR	1	1
0x40007404	Peripheral	DAC_SWTRIGR	1	1
0x40010000	Peripheral	AFIO_EVCR	1	0
0x40010004	Peripheral	AFIO_MAPR	1	0
0x40010400	Peripheral	EXTI_IMR	1	1
0x40010414	Peripheral	EXTI_PR	0	1
0x40013400	Peripheral	NOT IN SVD	1	1
0x40013800	Peripheral	USART1_SR	0	1

Address	Type	Name	Boot	Steady
0x40013804	Peripheral	USART1_DR	0	1
0x40020000	Peripheral	DMA1_ISR	0	1
0x40020004	Peripheral	DMA1_IFCR	1	1
0x40020030	Peripheral	DMA1_CCR3	0	1
0x40020058	Peripheral	DMA1_CCR5	0	1
0x40020400	Peripheral	NOT IN SVD	0	1
0x40020404	Peripheral	NOT IN SVD	0	1
0x40021000	Peripheral	RCC_CR	1	1
0x40021004	Peripheral	RCC_CFGR	1	1
0x40021008	Peripheral	RCC_CIR	1	0
0x4002100c	Peripheral	RCC_APB2RSTR	1	1
0x40021010	Peripheral	RCC_APB1RSTR	1	1
0x40021014	Peripheral	RCC_AHBENR	1	1
0x40021018	Peripheral	RCC_APB2ENR	1	1
0x4002101c	Peripheral	RCC_APB1ENR	1	1
0x40021020	Peripheral	RCC_BDCR	1	1
0x40021024	Peripheral	RCC_CSR	1	1
0x40022000	Peripheral	FLASH_ACR	1	1
0x4002200c	Peripheral	FLASH_SR	1	1
0x40022010	Peripheral	FLASH_CR	1	1
0x4002201c	Peripheral	FLASH_OBR	1	0
0x40022020	Peripheral	FLASH_WRPR	1	0
0x42200000	Peripheral	NOT IN SVD	1	0
0x42420000	Peripheral	NOT IN SVD	1	1
0x424200d8	Peripheral	NOT IN SVD	1	1
0x42420480	Peripheral	NOT IN SVD	1	0
0xa0000060	External_device	FSMC_PCR2	1	0

Address	Type	Name	Boot	Steady
0xa0000064	External_device	FSMC_SR2	1	0
0xa0000074	External_device	FSMC_ECCR2	1	0
0xa0000080	External_device	FSMC_PCR3	1	0
0xa0000084	External_device	FSMC_SR3	1	0
0xa0000088	External_device	FSMC_PMEM3	1	0
0xa000008c	External_device	FSMC_PATT3	1	0
0xa0000094	External_device	FSMC_ECCR3	1	0
0xa00000a0	External_device	FSMC_PCR4	1	0
0xa00000a4	External_device	FSMC_SR4	1	0
0xa00000a8	External_device	FSMC_PMEM4	1	0
0xa00000ac	External_device	FSMC_PATT4	1	0
0xe000e010	Private_peripheral_bus_Internal	STK_CTRL	1	1
0xe000e100	Private_peripheral_bus_Internal	NVIC_ISER0	1	0
0xe000e180	Private_peripheral_bus_Internal	NVIC_ICER0	1	0
0xe000e284	Private_peripheral_bus_Internal	NVIC_ICPR1	1	0
0xe000e400	Private_peripheral_bus_Internal	NVIC_IPR0	1	0
0xe000ed00	Private_peripheral_bus_Internal	SCB_CPUID	1	0
0xe000ed04	Private_peripheral_bus_Internal	SCB_ICSR	1	0
0xe000ed08	Private_peripheral_bus_Internal	SCB_VTOR	1	1
0xe000ed0c	Private_peripheral_bus_Internal	SCB_AIRCR	1	1
0xe000ed10	Private_peripheral_bus_Internal	SCB_SCR	1	0
0xe000ed24	Private_peripheral_bus_Internal	SCB_SHCRS	1	0
0xe000ed28	Private_peripheral_bus_Internal	SCB_CFSR_UFSR_BFSR_MMFSR	1	0
0xe000ed2c	Private_peripheral_bus_Internal	SCB_HFSR	1	0
0xe000ed30	Private_peripheral_bus_Internal	NOT IN SVD	1	0
0xe000ed34	Private_peripheral_bus_Internal	SCB_MMFAR	1	0
0xe000ed38	Private_peripheral_bus_Internal	SCB_BFAR	1	0

Address	Type	Name	Boot	Steady
0xe000edfc	Private_peripheral_bus_Internal	NOT IN SVD	0	1
0xe000ef00	Private_peripheral_bus_Internal	NVIC_STIR_STIR	1	0

REFERENCES

[1] B. Zhao *et al.*, "A large-scale empirical study on the vulnerability of deployed iot devices," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 3, pp. 1826–1840, 2020.

[2] *Ripple 20 - JSOF*, https://www.jsof-tech.com/disclosures/ripple20/, Accessed: 2024-07-07.

[3] *Zscaler ThreatLabz Finds 400% Increase in IoT and OT Malware Attacks Year-over-Year*, https://www.zscaler.com/press/zscaler-threatlabz-finds-400-increase-iot-and-ot-malware-attacks-year-over-year-underscoring, Accessed: 2024-07-07, Oct. 2023.

[4] N. Redini *et al.*, "BootStomp: On the security of bootloaders in mobile devices," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 781–798.

[5] H. Wen, Z. Lin, and Y. Zhang, "FirmXRay: Detecting Bluetooth Link Layer Vulnerabilities From Bare-Metal Firmware," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 167–180.

[6] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A Large-Scale Analysis of the Security of Embedded Firmwares," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 95–110.

[7] A. Costin, A. Zarras, and A. Francillon, "Automated dynamic firmware analysis at scale: a case study on embedded web interfaces," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016, pp. 437–448.

[8] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 480–491.

[9] Y. David, N. Partush, and E. Yahav, "Firmup: Precise static detection of common vulnerabilities in firmware," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 392–404, 2018.

[10] B. Feng, A. Mera, and L. Lu, "P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1237–1254.

[11] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for linux-based embedded firmware.," in *NDSS*, vol. 1, 2016, pp. 1–1.

[12] K. Cheng *et al.*, "DTaint: detecting the taint-style vulnerability in embedded device firmware," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, IEEE, 2018, pp. 430–441.

[13] Q. Li, X. Feng, R. Wang, Z. Li, and L. Sun, "Towards fine-grained fingerprinting of firmware in online embedded devices," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, IEEE, 2018, pp. 2537–2545.

[14] L. Zhu, X. Fu, Y. Yao, Y. Zhang, and H. Wang, "FIoT: detecting the memory corruption in lightweight IoT device firmware," in *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, IEEE, 2019, pp. 248–255.

[15] W. Zhou *et al.*, "Phantom Device Attack: Uncovering the Security Implications of the Interactions among Devices, IoT Cloud, and Mobile Apps," *arXiv. org*, 2019.

[16] N. Redini *et al.*, "Karonte: Detecting insecure multi-binary interactions in embedded firmware," in *2020 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2020, pp. 1544–1561.

[17] K. Koscher, T. Kohno, and D. Molnar, "SURROGATES: Enabling Near-Real-Time Dynamic Analyses of Embedded Systems," in *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015.

[18] E. Gustafson *et al.*, "Toward the analysis of embedded firmware through automated re-hosting," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, 2019, pp. 135–150.

[19] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang, and Y. Kim, "Firmae: Towards large-scale emulation of iot firmware for dynamic analysis," in *Annual Computer Security Applications Conference*, 2020, pp. 733–745.

[20] C. Spensky *et al.*, "Conware: Automated modeling of hardware peripherals," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021, pp. 95–109.

[21] J. Zaddach, L. Bruno, A. Francillon, D. Balzarotti, *et al.*, "AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares.," in *NDSS*, vol. 14, 2014, pp. 1–16.

[22] A. Fasano *et al.*, "SoK: Enabling security analyses of embedded systems via rehosting," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021, pp. 687–701.

[23] P. Subramanyan, S. Malik, H. Khattri, A. Maiti, and J. Fung, "Verifying information flow properties of firmware using symbolic execution," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2016, pp. 337–342.

[24] E. Johnson *et al.*, "Jetset: Targeted firmware rehosting for embedded systems," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 321–338.

[25] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution," in *22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 463–478.

[26] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware.," in *NDSS*, vol. 1, 2015, pp. 1–1.

[27] *A powerful disassembler and a versatile debugger*, https://hex-rays.com/ida-pro, Accessed: 2024-07-07.

[28] M. Jiang, Y. Zhou, X. Luo, R. Wang, Y. Liu, and K. Ren, "An empirical study on arm disassembly tools," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 401–414.

[29] *Hopper Disassembler, the reverse engineering tool that lets you disassemble, decompile and debug your applications.* https://www.hopperapp.com, Accessed: 2024-07-07.

[30] V. Salis, T. Sotiropoulos, P. Louridas, D. Spinellis, and D. Mitropoulos, "Pycg: Practical call graph generation in python," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, IEEE, 2021, pp. 1646–1657.

[31] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Nautilus: Fishing for deep bugs with grammars.," in *NDSS*, 2019.

[32] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring Network Structure, Dynamics, and Function using NetworkX," in *Proceedings of the 7th Python in Science Conference*, G. Varoquaux, T. Vaught, and J. Millman, Eds., Pasadena, CA USA, 2008, pp. 11–15.

[33] O. Tange, "Gnu parallel - the command-line power tool," *;login: The USENIX Magazine*, vol. 36, no. 1, pp. 42–47, Feb. 2011, Accessed: 2024-07-07.

[34] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of executable code revisited," in *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, IEEE, 2002, pp. 45–54.

[35] P. Huitsing, R. Chandia, M. Papa, and S. Shenoi, "Attack taxonomies for the modbus protocols," *International Journal of Critical Infrastructure Protection*, vol. 1, pp. 37–44, 2008.

[36] *Modbus Application Protocol Specification*, https://modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf, Accessed: 2024-07-07, Apr. 2012.

[37] *Monolithic firmware collection*, https://github.com/ucsb-seclab/monolithic-firmware-collection, Accessed: 2024-07-07, 2022.