PinDown:
Generalized Application Code Identification And Functional Component Analysis In RTOS-based Firmware

D.A. Prinsze

TU Delft

PinDown: Generalized Application Code Identification And Functional Component Analysis In RTOS-based Firmware

by

D.A. Prinsze

to obtain the degree of Master of Science in Computer Science
at the Delft University of Technology,
to be defended publicly on 7^{th} of October, 2024

Student number:	4346106	
Graduation committee:	Georgios Smaragdakis,	TU Delft
	Alex Voulimeneas,	TU Delft
	Jérémie Decouchant,	TU Delft
	Andrea Continella,	UTwente

TU Delft

Preface

This thesis analyzes the effectiveness of identifying and leveraging RTOS components within RTOS based firmware in order to identify application code. This project was performed at the Cyber Security Group at the technical university of Delft in conjunction with the university of Twente. The supervising professor is Georgios Smaragdakis, a professor of Cybersecurity at TU Delft. The assisting supervising professor is by Andrea Continella, an associate professor at the university of Twente. The responsible professor is Alex Voulimeneas, an assistant professor at the Cyber Security Group at TU Delft. The final committee member member is Jérémie Decouchant, an assistant professor from the Distributed Systems Group at TU Delft.

The source code will be made publicly available at a later date[1].

[1] https://github.com/utwente-scs/firmware-decomposition/

Acknowledgments

I want to thank my family and friends for their patience and support during this period. Most of all, I want to thank my wonderful partner for her encouraging words and endless kindness during this chapter of my academic career. Furthermore, I want to thank my supervisors from Delft and Twente for their tremendous support and words of encouragement whenever I felt at a loss. I had a lot of fun working on this project and look forward to the next phase of my life.

D.A. Prinsze
Delft, October 2024

Abstract

Small embedded devices are becoming more prevalent in the world with each passing year to improve our quality of life. However, as more devices are created, an increasing number of older devices are declared obsolete despite still being used. This results in an increasing amount of devices being vulnerable to exploitation due to a lack of security updates. Identifying these vulnerabilities manually without any system knowledge is an arduous task, and current state-of-the-art technologies do not perform generalized analysis in RTOS-based firmware. In this work, we present PinDown, an analysis framework that enables the automated identification of application code in RTOS-based firmware without requiring partial system knowledge. By identifying functions that modify the heap, we can identify RTOS components that can be leveraged to locate memory regions that host application code.

Contents

1 Introduction

Electronic devices play a critical role in our society. It is estimated that nowadays there are about 17.08 billion connected *Internet of Things* (**IoT**) devices with statistics indicating that this amount will reach a total of 32.1 billion by 2030, nearly doubling within six years [1]. Consumer electronics represent a large subsection of these devices as people have an ever-increasing need for control over home appliances such as indoor climate control, being notified of who is at their front door, or turning on their lights from a remote location. Aside from consumer electronics, IoT devices also play a large role within *critical infrastructure* (**CI**) such as water management facilities, power distribution stations, public infrastructure management systems, the medical domain as well as the financial sector [2][3][4]. Within CI these devices are often deployed as advanced sensors, which are responsible for relaying information to central control hubs that depend on the integrity of these devices to fulfill their function. Due to the broad deployment of such devices in our society, the attack surface has increased considerably. This can mainly be attributed to the fact that these devices and the protocols they utilize often introduce vulnerabilities, as established in two separate security reviews by Nadir et al. [5] and Pliatsios et al. [6]. Several attacks on CI in the past were enabled by the utilization of insecure inter-device communication protocols, lack of adherence to established security regulations, and the absence of updating or patching of these critical devices [7]. This makes CI a high-priority target for actors who aim to disrupt and damage society or to perform espionage. As such it is paramount to secure these devices and to inspect any suspect devices for potential vulnerabilities to prevent such damage from being realized.

The need to secure potentially vulnerable CI devices against malicious actors is high. Frank Ebbers has shown that many security vulnerabilities can persist because of lacking incentive to update the firmware on IoT devices [8]. Ebbers analyzed connected IoT devices on the Internet and established that devices that ran the most up-to-date firmware only represented 2.45% of all analyzed devices. Furthermore, they determined that the average time since a prior update was performed is 19.2 months. The analysis also indicates that device type and manufacturer are the most significant factors in determining the likelihood of whether or not firmware will be updated. The inability to update firmware with the goal of patching security vulnerabilities depends on several factors, such as the logistics of performing updates, whether a downtime of the device in question is acceptable, and whether the manufacturer still supports the device. The analysis by Frank Ebbers shows that manufacturers tend to prioritize the development of new commercial products over maintaining older devices, allowing various security vulnerabilities to persist. The lack of support from device manufacturers means we must analyze and secure vulnerable devices ourselves.

A characteristic of many IoT devices that can be analyzed and studied is the operating system or the operating system libraries that provide a form of abstraction to the device. Most firmware utilizes some implementation of a *real time operating system* (**RTOS**) in order to facilitate developers and make development for devices easier and faster. Such RTOSs provide a *hardware abstraction layer* (**HAL**), which allows on-board peripherals to be invoked or addressed. Firmware that is built on top of an RTOS (or libraries of an RTOS) is referred to as **Type-II** or **Type-III** firmware [9][10]. Several operating systems for embedded devices are open source and maintained by a large collective of developers.

An example of such an RTOS is RIOT OS[1]. All the source code for RIOT is available online and can be modified by anybody who wishes to do so. Such open-source frameworks enable developers to adjust certain functionalities and tailor them for their specific use case. This form of development also allows people to submit adjustments and other modifications for approval to incorporate them into the framework that is then made available to others. Such changes need to be approved by the respective maintainers of the project, mitigating the possibility of erroneous or malicious code being incorporated. This transparency allows individuals to inspect such frameworks and identify bugs, which can then be reported to the maintainers of the project, ensuring that a collective of interested parties can aid in securing such frameworks. Using these publicly available open-source frameworks provides several benefits to embedded developers as well. When developers opt to use an established framework such as RIOT, they can reasonably assume that the framework is secure and can utilize the well-documented functionalities in order to develop their custom application. As such, they only need to be concerned with developing their application and using all the tools that are available.

Contrary to identifying user application code, analyzing the RTOS frameworks is less challenging due to the open-source nature of these systems since the translation to firmware usually conforms to a one-to-one conversion, assuming that no modifications were made. However, when developers decide to strip such a system of bloating components that are not utilized, this notion falls apart as we can no longer be certain that we are comparing the correct components. This is because binary firmware images are generally distributed in a custom format that has been stripped, leaving it devoid of debugging symbols and function names. This results in a lack of discerning factors that enable efficient manual analysis. While these frameworks provide improvements for development, they also make it more challenging to identify crucial components that are more likely to introduce vulnerabilities. These frameworks introduce several hundreds, if not thousands, of functions. Of all these functions, only a few host the user application code, which can be interacted with, whilst the rest have a mostly supportive role in executing those functionalities. The sheer complexity of these binary images, in conjunction with the lack of context for each function thanks to the custom format, all pose an immense challenge when we want to discern between user applications and framework-supporting functions.

One of the prevalent ways of analyzing IoT devices is employing firmware analysis. The process of the analysis entails that one acquires a binary image of the firmware on a target device and reverses the implementation of the program code in order to establish its function and identify potential vulnerabilities that enable exploitation [11][12][13][14]. However, such manual analysis of IoT devices with the intent to secure them has proven to be a difficult and time-consuming effort. As such, research efforts have focused on automated analysis, vulnerability identification, and mitigation in various ways. Examples of recent research efforts are state-of-the-art technologies such as ArgXTract [15] and ArmPatch [16]. These solutions focus on analyzing and patching security vulnerabilities in devices that lack proper support from the manufacturer. These solutions use device characteristics to identify vulnerabilities that could be exploited (such as buffer overflows). After identifying such a vulnerability, their solution introduces new code into the binary to prevent exploitation and mitigate the vulnerability. The ability to perform such analysis and patch whatever vulnerability is found is of utmost importance in the absence of regulations that guarantee the security of IoT devices. However, the field of firmware analysis is challenging given the firmware's complexity and the myriad of hardware configurations available, making it hard to find a "one size fits all" solution.

In the field of automated analysis, current state-of-the-art solutions focus mainly on analyzing every single function in a binary in search of common types of vulnerabilities through static analysis. These common vulnerabilities often focus on memory-related bugs, which can be identified through symbolic execution. This approach is, however, not optimal due to the nature of static analysis and how certain bugs require the environment to be fully initialized before they become apparent. Furthermore, static analysis also tends to identify vulnerabilities that can not be exploited in practice. However, dynamic analysis is often not feasible either, as simulating all the code in the firmware does not guarantee that vulnerable sections within the firmware will be reached. There is a need for analysis techniques that perform a more targeted analysis of relevant sections within the firmware, namely the user application code, without requiring partial knowledge. As developers create custom applications on top of RTOS frameworks, these applications are more likely to contain undiscovered vulnerabilities.

In order to streamline this process of identifying user application code, we require a generalized analysis approach that can leverage characteristics that are generally present by design regardless of

[1]RIOT: `https://github.com/RIOT-OS/RIOT`

hardware configuration, RTOS, or any possible modifications made without partial knowledge or being aware of the presence of any of the factors as mentioned earlier.

1.1. Research questions

Whilst there are state-of-the-art solutions aimed at deploying patches in an automated manner or examining firmware in search of common vulnerabilities, no solution exists that aims to identify user application-related functions within firmware built on top of an RTOS. The identification of specific application code-related components within an image would aid in the manual analysis of firmware as we only focus on the relevant parts of a binary and enable the deployment of patches in a targeted manner, accelerating the entire process of analysis. Furthermore, the ability to identify application code would help uncover vulnerabilities and aid in establishing the exact functionality of a device and the library components the application utilizes to enable its function. In order to address and resolve this knowledge gap, we address the following research questions in this work:

RQ1. *How can we identify functional components in monolithic firmware?*

In order to address this main research question, we answer the following sub-questions:

SQ1. *What are the prevalent mechanisms or structures that launch or initiate application code in real time operating systems?*

SQ2. *How can these mechanisms or structures be leveraged in order to allow for effective application code discovery?*

SQ3. *To what extent can this procedure be generalized in a way such that we can analyze firmware based on different RTOSs?*

1.2. Contributions

In this thesis, I present **PinDown**[2], a fully automated analysis tool that utilizes static analysis and symbolic execution in order to accurately identify functional components and application code within RTOS-based firmware. PinDown effectively identifies application code within firmware hosted on an ARM Cortex-M processor regardless of RTOS. The analysis that PinDown performs is based on static analysis and symbolic execution, which makes the implementation more efficient than other dynamic analysis approaches, as it is scalable. PinDown statically performs all analysis by leveraging the `angr`[3] framework, which enables the manipulation of variables and the discovery of relations between functions through control-flow graph analysis. Furthermore, the entire analysis process is fully automated, meaning that one only needs to provide the tool with a binary firmware image as PinDown establishes information such as the binary's entry point and its base address offset by itself. Following the identification of the base address is an in-depth analysis of the binary firmware image, in which the addresses of the application code and functional components are discovered. After the analysis is complete, PinDown generates candidate function addresses within the binary that we label as functional components and application code.

PinDown performs its analysis in three separate stages. Pindown will start with the first stage of analysis, in which functions that implement thread creation are identified. The second stage of analysis performs analysis over argument values, which are identified at the call sites that target these functions. The third and final stage of analysis is the analysis of the arguments that represent function addresses and their implementation to determine the locations of user application code. This approach was constructed because we discovered a strong relationship between the creation of threads and the invocation and execution of user applications within RTOS-based firmware. This perceived relevance was discovered during an analysis of several different RTOSs and RTOS-based firmware images, which all implement thread creation to directly invoke a user application or set up a component, which proceeds to further initialize the system and then invoke a user application.

We verify the correctness of the approach through experimental validation, which consists of applying the tool to a manually generated dataset of RTOS-based firmware and ”wild” firmware images (i.e., firmware extracted from real-life devices). When using PinDown on the generated dataset to establish a ground truth, it could distill the several hundreds of functions within a binary image down to an average of four if at least the second analysis phase was successful. The binary firmware images in

[2]PinDown source code: `https://github.com/utwente-scs/firmware-decomposition`
[3]`angr` framework: `https://angr.io/`

the dataset that was used to establish the ground truth of the project are generated using six different RTOS at varying optimization levels ranging from O0 to O5. Afterward, several databases containing monolithic firmware[4,5] were searched for wild firmware which was compiled for Cortex-M based devices. PinDown then analyzed all these different images to identify the application code, the correctness of which was manually verified by reverse engineering. Of all identified function candidates in these binaries, the function address that contained the application code was found in the majority of all test cases for both the ground-truth dataset and the wild firmware. However, other functions were sometimes marked as potential candidate functions, resulting in an average false positive rate of about 50.26% while reducing several hundreds of candidate functions by 99.18% on average. If PinDown could identify potential candidates, then we should still manually verify which of these functions performs the tasks of the device. Luckily, manual verification has proven to be quite easy, as relations between candidates will quickly indicate the addresses of the application code based on how this functionality is usually compartmentalized. PinDown was also used to analyze binaries that had not been previously inspected from several datasets that had been utilized in related work.

In summary this work has provided the following contributions:

- An analysis of task initialization processes in firmware;
- An analysis of application initialization techniques within real-time operating systems;
- A generalized approach of identifying application code within RTOS based firmware;
- Provide an extension to existing state-of-the-art technology in order to identify user applications;
- The PinDown framework itself.

1.3. Outline

The rest of this work is structured as follows. Chapter 2 contains background information upon which the research was based as well as elaboration upon concepts that are required for understanding this work.

Chapter 3 goes more in-depth on the current state-of-the-art research and techniques employed in this work. A summary of the most important works for this thesis is also provided.

Chapter 4 contains the construction of the approach and provides supportive arguments as to why the approach is logically sound. Furthermore, a high-level overview of the approach is provided, and the research constraints are further elaborated upon.

Chapter 5 contains all the results of the experimental validation as well as discusses the experimental setup, how the dataset was acquired, and an in-depth analysis of the performed experiments.

Finally, chapter 6 contains the conclusion of the research, the uncovered limitations, and possible suggestions for future work.

[4] Monolithic firmware dataset: `https://github.com/ucsb-seclab/monolithic-firmware-collection/tree/master`

[5] RTOSExploration database: `https://github.com/RTOSExploration/lctes2023-artifact/tree/main/artifact/bitcode-db`

2

Background

This chapter provides more information about important concepts relevant for understanding the final work. We assume a basic understanding of operating systems, reverse engineering, and analysis techniques as we will build our implementation on top of existing knowledge. Section 2.1 provides more information about the classes of techniques that are utilized in the field of reverse engineering. Furthermore, this section provides a comparison of the two classes to highlight the benefits and downsides of each respective class. Section 2.2 provides information about the different kinds of firmware, their characteristics as well as how these can be categorized into separate groupings. Section 2.3 provides an analysis of several different RTOSs which were chosen based on popularity in the industry. The results and insights of this analysis were utilized to construct a general approach to aid in the identification of user applications. Section 2.4 provides information about an analysis technique that was leveraged in order to identify critical components within a binary firmware image. Lastly, section 2.5 defines the scope of the work as well as how we came to the conclusion that this scope is appropriate.

2.1. Security analysis of IoT devices

Embedded and *internet of things* (**IoT**) devices are devices that can have a major impact on the fields in which they are deployed if compromised, making them high-value targets for malicious actors. While this is especially true for critical infrastructure that utilizes these devices, it also holds for devices targeted at consumers. As such, it is important to understand the security risks associated with such devices and where and how vulnerabilities manifest themselves. Furthermore, we need to understand how we can analyze firmware in search of vulnerabilities and what techniques are available.

A study performed by Nadir et al. [5] found that there are two primary ways of identifying vulnerabilities in firmware: through fuzzing/fuzz testing and by applying static or dynamic analysis (or a mix thereof). Fuzzing typically entails the generation of system inputs with the intent to trigger events that lead to system exploitation. Several works focus on the usage of fuzzing in order to uncover vulnerabilities. An example of such a framework is IoTFuzzer, developed by Chen et al. [17]. This framework utilizes fuzzing to find memory vulnerabilities in IoT devices by dynamically relating crashes to certain inputs. A downside is that we may be unable to reach critical code sections utilizing dynamic analysis. As such, fuzzing does not guarantee the total coverage of an IoT device. Another downside is that we rely on possessing the physical device or being able to simulate the device in question, all of which introduce overhead and reduce the scalability of applying fuzzing techniques to IoT devices. Due to this overhead, most solutions proposed in the field of vulnerability analysis are based on static and/or dynamic analysis techniques.

2.1.1. Dynamic analysis

Dynamic analysis is closely related to fuzzing but focuses on analyzing properties associated with executing a program. Generally, the benefit of dynamic analysis is the high degree of accuracy that is provided, as no approximation or abstraction is provided. This is because in dynamic analysis, the exact run-time behavior is analyzed, reducing the uncertainty of which control flow paths were taken, the computed values, or how long the program took to execute [18][19]. Furthermore, we do not need

to decompile or decrypt program code; we simply execute the program code and observe its behavior. As a side effect of program execution, we can observe any vulnerabilities if the appropriate event is triggered. Another benefit of dynamic analysis is that the run-time execution could provide insight into the program code that is being executed.

However, dynamic analysis does pose some challenges. Dynamic solutions do not scale as well as several static analysis techniques do. Generally, dynamic analysis is performed by investigating the device and observing its function. Such analysis could only be scaled through virtualization. When utilizing virtualization, it could be that the program code that requires analysis cannot be virtualized due to limitations. Furthermore, such virtualization introduces a lot of overhead and requires a lot of computational resources before it can be effective [5].

2.1.2. Static analysis

Static analysis entails the analysis of IoT devices without the need to execute the software present in such devices. This means that if you can acquire a binary firmware image from a device, you can analyze these images to identify the issues within the firmware without having access to the physical device. This implies that static analysis is ideal regarding scalability if automated. On the contrary, manual static analysis has been known to be a costly and time-consuming endeavor. Furthermore, static analysis has proven to be most effective when we want to improve existing security features and are not required to bypass encryption or obfuscation [5]. Another benefit of static analysis is that we can analyze memory regions that can not be reached through dynamic analysis. Even though the system will never use these regions, the implementation of functions in these regions still presents a potential vulnerability as potential exploits could leverage these regions by means of *return oriented programming* [20] as they could be incorporated in the *ROP-chain*. An example of a work that relies on static analysis to identify vulnerabilities in multi-binary systems is KARONTE by Redini et al. [11]. The main motivation for this research was based on the fact that non-standard configurations in IoT devices left current static and dynamic analysis techniques ineffective. This is because many existing devices implement their function through the usage of multiple binaries that interact with each other. KARONTE was developed to provide an effective analysis approach in such multi-binary systems, which is done by tracking and modeling interactions between binaries. As information is tracked between binaries through taint-analysis, the amount of false positives is reduced, which is one of the known downsides of static analysis as shown in figure 2.1. KARONTE has been shown to work very well on Linux-based firmware, having found multiple previously unknown zero-day vulnerabilities. However, no such approach exists yet for RTOS-based firmware.

We want to construct a generalizable approach that can be applied to any binary firmware image based on an RTOS framework. It becomes clear that we need to consider utilizing existing tools and frameworks such as `angr`[1] that enable static analysis given how need to be able to analyze binaries regardless of their implementation and functionality. We cannot make assumptions about the dynamic functionality of binary firmware images, and as such, we cannot derive a way to identify user applications using such techniques. As such, we need to base our approach on static analysis and accept the possibility that our approach will have a higher rate of false positives.

2.2. Embedded Operating Systems

An embedded or *internet of things* (**IoT**) device generally has an integrated circuit that contains all the program code the device needs in order to fulfill its function. Such program code is called *firmware*.

Firmware comes in many different forms and configurations, which makes it hard to divide them up into categories that truly encompass all functionality whilst enabling comparative studies. As our research focuses on the construction of a supporting framework in the analysis and identification of application code, we will use the classifications utilized by Muench et al. [10] and Qasem et al. [9]. We have opted for this classification structure because we aim to leverage operating system-related mechanisms in order to discover user applications. This also made sense since we want to discover user application code within frameworks to enable targeted analysis and patching. The utilized classification structure also provides information about possible security mechanisms that can be provided based on the type of classification. The classification system splits the different types of firmware into three separate groups: Type-I, Type-II, and Type-III.

[1] `angr` framework: `https://angr.io/`

Discipline	Static	Dynamic
Methodology	Analysis performed by statically analyzing code	Behavior of program is observed by executing it
IoT Device	Not required for static analysis	IoT device is required for dynamic analysis
Scalability	Scalable solutions can be developed if large scale of firmware repository is available	Scalable solutions cannot be achieved without virtualization
Metadata	Desired but not necessary	Necessary
Virtualization	Not required	Emulation may fail even if firmware and metadata is available due to multiple reasons (see section 5.5)
Decompilation	1. Not possible due to unavalability of sepecific decompilers 2. Decompilation is not accurate	No need for code decompilation in dynamic analysis
Runtime insights	Does not provide insight about the code/data or complex problems	Can provide insights about code/data while executing
Unused code	Possible to analyze unused code	Not feasible to analyze unused code
False positives	Higher rate of false positives	No issues of false positives
Vulnerability detection	Can identify vulnerabilities like memory-corruption, segmentation faults, uninitialized variables etc.	Can detect any vulnerability when code is executed
Firmware acquisition	Firmware acquisition is necessary for static analysis	No need to acquire firmware if actual device is (remotely) available
Runtime issues	Static analysis cannot capture runtime vulnerabilities	Dynamic analysis can capture runtime vulnerabilities
Encryption/Obfuscation	Hinders static analysis	Not an issue for dynamic analysis
Unexploitable code	Static analysis can find vulnerabilities that are unexploitable	Cannot find unexploitable code

Figure 2.1: A table constructed by Nadir et al. [5] that highlights the benefits and downsides of static analysis compared to dynamic analysis.

- *Type-I: General purpose OS-based.* Type-I firmware is often defined as being built on top of a modified or lightweight version of an existing operating system. Typically, such operating systems are based on Linux, and they are usually designed to handle complex logic and enable networking and internet services. The benefit of building on top of such an operating system is that it allows for easier integration of existing UNIX software suites that can run in POSIX environments such as BusyBox[2] and micro-C[3].
- *Type-II: Embedded OS-based.* Type-II firmware is built for devices that are constrained in computational resources and power and are generally built on top of a minimal operating system. Generally, Type-II firmware is built on top of hardware that does not possess a *memory management unit* (**MMU**) as is the case in Cortex-M-based hardware. Generally, the firmware for such devices still contains a logical separation between the application and the kernel. An example of an OS in Type-II firmware is Zephyr.
- *Type-III: No OS abstraction* Type-III firmware is characterized as not having an operating system. Instead, the software depends on a single control loop and interrupt handler to respond to interactions with the device's peripherals. Such firmware can be completely customized or based on existing operating system libraries. These libraries provide a form of abstraction, but the eventual firmware that is compiled will still contain system and application code compiled together as monolithic software. Contiki and mBed OS provide such libraries.

In this work, we mostly focus on the analysis of firmware that is built on top of a *real-time operating system* (**RTOS**) or that utilizes existing RTOS libraries. An RTOS is defined by the need for the timely execution of tasks and the processing of data necessary to perform those tasks. This makes such RTOSs ideal for the development of applications intended for IoT devices, as these typically depend on the efficient execution of their intended function. While current works focus on analyzing binary interactions within Linux-based and bare-metal devices, there is no work that aims to address the generalized analysis of RTOS-based firmware. This could be due to the lack of an effective way of categorizing firmware into classes. Instead, other works classify firmware based on implementation and features present in firmware images. In this work, we aim to create a generalizable approach for identifying user application code within Type-III firmware that was built using RTOS libraries, as well as

[2]BusyBox: `https://www.busybox.net/`
[3]uClibC: `https://uclibc.org/`

Type-II firmware that was built on top of a single-purpose RTOS.

2.3. RTOS analysis

We need to analyze and reverse engineer a group of real-time operating systems to find overlapping structures or mechanisms that can be leveraged to construct a generalizable approach. Such analysis is required in order to get an idea of how applicable the approach will be to any real-time operating system. For this analysis, we have chosen six popular open-source operating systems[4] that are widely used in the industry. Whether an operating system is popular is determined based on its ranking on Github, and additional choices are based on the availability of wild firmware for the experimental validation. The following real-time operating systems were chosen for analysis:

- LittleKernel[5]
- RIOT[6]
- mBed OS[7]
- Zephyr[8]
- Contiki-NG[9]
- NuttX[10]

A set of firmware images is compiled for each operating system at varying optimization levels, ranging from no optimization to O5. This was done to uncover functions that were abstracted away and estimate whether this required to be accounted for in later analysis phases (see section 4.5).

Ghidra[11] is a reverse engineering tool developed by the NSA that disassembles executables as well as binaries. Given how we have manually compiled a set of binaries, we have the corresponding *executable and linkable format* (**ELF**) files as well that can be disassembled by Ghidra. These files contain information about function names, the base address offset, and other otherwise absent characteristics. Real-time operating systems require a way of handling whatever user application code needs to be executed by managing threads. Such operating systems usually implement a function that creates a thread whenever a process needs to be called and returns a corresponding unique identifier for this thread. Generally, thread creation involves generating a new stack and allocating space on the heap for local variables of the accompanying function that it calls [21]. This means that any application that is required to run in parallel to whatever functions the operating system is performing will most likely be instantiated in this manner. Analyzing the aforementioned operating systems by investigating functions whose task is to create threads shows that these functions play a critical role in the initialization of user applications. While the implementation of these functions is different for each operating system, analysis shows that each such function has been found to call application code directly or is responsible for initializing another process, which initializes such code.

The following schematics indicate how these functions relate to the process of initializing application code. The figures shown in 2.2, 2.3, 2.4, 2.5, 2.6 and 2.7 provide a schematic overview of the relation between thread creation and user application code for LittleKernel, MBedOS, NuttX, RIOT, Zephyr and Contiki-NG respectively. Given the entry-point function of a binary f_n, this function may contain function calls to other targets where n is the layer in the call graph starting at f_0. A function that is tasked with the creation of threads is labeled TC. As stated earlier in this section, TC functions require another function as an argument to initialize the stack and whatever local variables such a function needs. We denote a function passed as an argument as sf_n, where n denotes the layer in the call graph from the point where it was passed as an argument. The analysis shows that typically, these functions are either a component that performs the initialization of the application code or the application code itself. When we find a component whose task it is to initialize application code, we find that there are several possible implementations of how this is achieved:

[4] Popular RTOS: `https://www.osrtos.com/`
[5] LittleKernel github: `https://github.com/littlekernel/lk`
[6] RIOT OS github: `https://github.com/RIOT-OS/RIOT`
[7] mBedOS github: `https://github.com/ARMmbed/mbed-os`
[8] ZephyrOS github: `https://github.com/zephyrproject-rtos/zephyr`
[9] Contiki github: `https://github.com/contiki-ng/contiki-ng`
[10] NuttX github: `https://github.com/apache/nuttx`
[11] Ghidra: `https://github.com/NationalSecurityAgency/ghidra/releases`

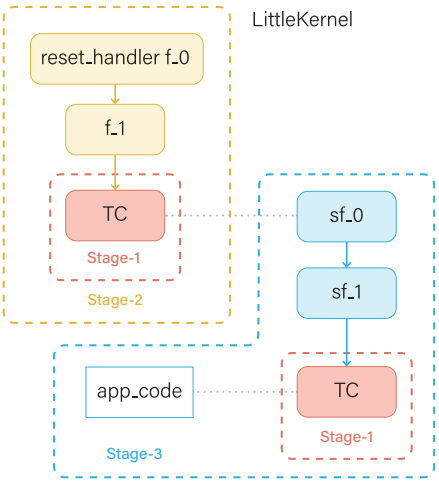

Figure 2.2: A schematic overview of the relation between thread creation and initializing application code in LittleKernel based firmware.

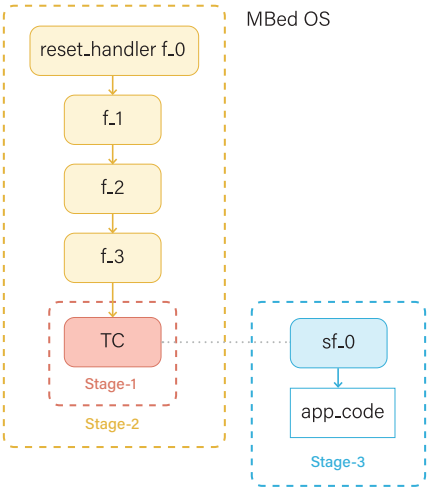

Figure 2.3: A schematic overview of the relation between thread creation and initializing application code in MBedOS based firmware.

1. Threads are created before a direct call is made to the application code where the call site only contains a single `bl` instruction in its block.
2. Threads are utilized in order to launch application code, which can be a single application or several applications that are launched individually through a loop where the addresses are typically stored in a list
3. There is no usage of thread creation, but several smaller functions are called which perform some initialization of the system or hardware

Based on these groupings, we can assign each operating system to a group that best describes its implementation. We find that Zephyr generally fits the description of group 1 and group 2 as Zephyr can launch single applications or uses thread creation to launch several applications. The general construction of LittleKernel-based firmware indicates that LittleKernel best fits the description of group 2 as no other initialization mechanisms are present, and every application is launched in a similar manner. MBed, Contiki, and RIOT all match the description of group 3 as shown in figures 2.3, 2.7, and 2.5, respectively. This analysis shows that we should be able to leverage functions that create threads to uncover application code in firmware based on the chosen operating systems. While this analysis has been performed on a very specific subset of all available operating systems, the notion that these systems require functions that create threads to manage whatever application is run on top of the system should be extendable to other systems. We base this on the idea that we are leveraging aspects that fit into basic operating system design principles.

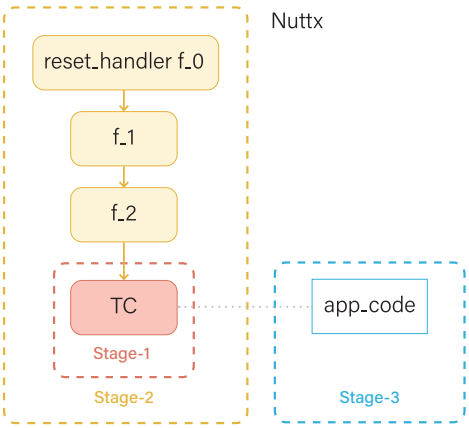

Figure 2.4: A schematic overview of the relation between thread creation and initializing application code in NuttX based firmware.

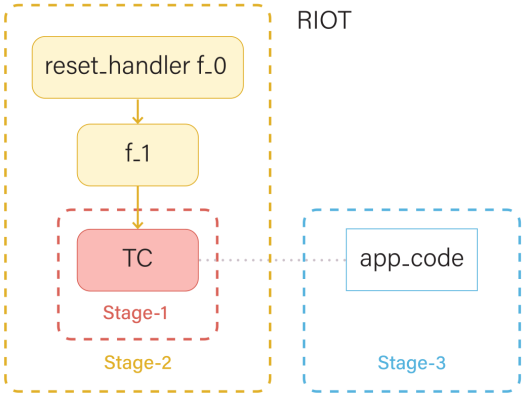

Figure 2.5: A schematic overview of the relation between thread creation and initializing application code in RIOT based firmware.

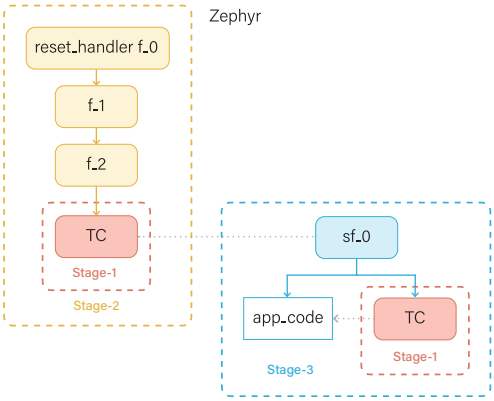

Figure 2.6: A schematic overview of the relation between thread creation and initializing application code in Zephyr based firmware.

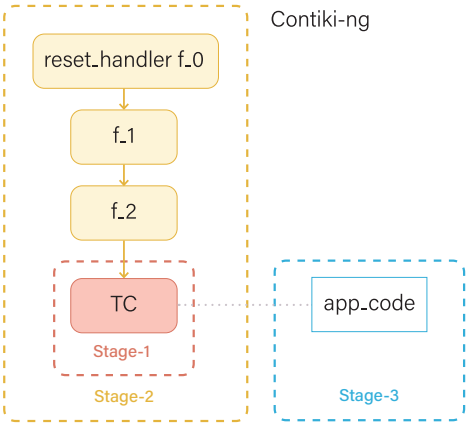

Figure 2.7: A schematic overview of the relation between thread creation and initializing application code in Contiki-NG based firmware.

2.4. Reaching definitions analysis

In this work, we aim to identify the user application code, and we have determined that its invocation is strongly related to functions that implement the creation of threads. As such, we need to identify an appropriate method of identifying these functions in a generalized manner. This means that we need to identify an overlapping component that is present in all implementations of such functions. The field of function discovery seems to focus mostly on how functions can be differentiated from data in binary code. This is useful as methods that aim to statically translate binaries must be aware of all the functions within a binary to successfully retarget it for another platform. Two works by Chen et al. highlight both the importance of such techniques as well as the applicability [22][23].

However, we need to identify functions based on their functionality and assume the function in question is one of many candidates. In our analysis of RTOSs, we determined that functions that implement thread creation depend on some form of heap modification to accommodate whatever function is passed to it as an argument. This means that if we can determine the function addresses of heap modifying functions, we can utilize *reaching definitions* (**RD**) analysis to identify candidates that depend

on heap modifying functions.

The concept of reaching definitions is inherent to the field of compiler construction theory. When a value is assigned to a variable, this is called a *definition*. We say that a definition d is able to *reach* a point p in the control-flow graph if there exists a path a between d and p where d is not overwritten [24]. As such, we are able to inspect reaching definitions by means of data-flow analysis. This analysis can be used to reach certain parts within the program code. In order to identify functions that implement thread creation, we can inspect the register values of heap modifying functions. We can then apply reaching definitions analysis to these variables up to the point where they were first declared.

We have determined that the initialization of application code depends on the usage of functions that implement thread creation and that functions that implement thread creation rely on heap modifying functions. This means that functions that implement thread creation will initialize variables that are associated with function size and function-specific variables. When we apply reaching definitions analysis to these variables, we can reach a function that has implemented thread creation to accommodate potential user application code. In order to do so, a *definition graph* is constructed. In such a graph, each node represents a register definition, and each edge represents the dependencies between different definitions. The following equation gives a formal definition of this data-flow analysis:

$$IN_n = \bigcup_{p \in pred(n)} OUT_p \tag{2.1}$$

$$OUT_n = GEN_n \bigcup (IN_n - KILL_n) \tag{2.2}$$

In equation 2.1 and 2.2, IN_n and OUT_n represent the incoming and outgoing information flows. In equation 2.2, GEN_n and $KILL_n$ respectively contain information flows that are newly generated and overwritten (killed). These sets are properties of each node in the definition graph and are expressed as follows:

$$GEN_n = \{\langle x, n\rangle : n \text{ def } x\} \tag{2.3}$$

$$KILL_n = \{\langle x, n'\rangle : n \text{ def } x, n' \in N\} \tag{2.4}$$

Equation 2.3 shows that a node generates a definition for each variable it defines, and equation 2.4 shows that a node kills a definition if one already exists.

This work implements reaching definitions analysis in order to uncover function addresses that implement thread creation by applying this analysis to basic heap modifying functions.

2.5. Constraints and scope

This research focuses on RTOS-based firmware, the target architecture of which is ARM Cortex-M. As the goal is to find a generalized approach for identifying application code within such binary firmware images considering the allotted time frame, it is necessary to first verify the existence and feasibility of such an approach. As bare-metal firmware does not contain any form of operating system that provides an abstraction, it becomes inherently difficult to differentiate between application code and functions that are tasked with handling events [9]. As such, the characteristics that could be utilized to discern between such components are ambiguous at best. RTOS-based firmware, on the other hand, contains a form of operating system, either general purpose or embedded, and, as such, it provides a separation that is potentially easier to identify. Due to the presence of an operating system, such firmware is more likely to contain indicators that enable the construction of a generalized approach. The lack of ambiguity in the layers of such firmware allows for leveraging principles from the operating system development field. This enables the construction of an appropriate strategy, reducing the need to rely on device-specific implementations, which can vary greatly between different hardware configurations.

As small IoT devices show the greatest need for targeted function analysis and patching, given the widespread deployment of such devices in critical infrastructure, we need to identify the type of hardware such devices typically require. Several papers, such as the analysis provided by Pinto et al.[25] and argXTract implementation by Sivakumaran et al.[15] corroborate the statement that the ARM platform is widely used by small and energy-efficient IoT devices. Furthermore, their analysis shows that ARMv7 Cortex-M4 is generally still widely considered the industry standard. While hardware is slowly transitioning to ARMv8, which provides more features in order to enable secure application development, as elaborated in chapter 2, many older devices would need to be replaced, which is a

costly endeavor. As the Cortex-A series plays a smaller role in terms of industry utilization, we will limit the scope to focus on ARM Cortex-M-based devices specifically, as these devices are more common and are less likely to be replaced.

An automated and generalizable approach requires that all necessary information for analysis can be acquired by means of deploying state-of-the-art analysis techniques. As ARM Cortex-M devices are required to be energy efficient and have several computational limitations, these devices tend to contain RTOS-based firmware as there is simply no need for all auxiliary functionality that more advanced operating systems contain. For these reasons, the research in this work will be limited to RTOS-based firmware where its target architecture is ARMv7 Cortex-M.

3

Related Work

In this chapter, we discuss the works and state-of-the-art technologies related to this work in more detail. We also discuss the reasoning behind adopting certain technologies. Section 3.2 goes in-depth about several works that have provided methods of deriving information about binary firmware images, such as the base address and instruction set architecture, without relying on partial knowledge. Section 3.2 discusses the techniques used in state-of-the-art technology to uncover specific functions and how this is performed. Section 3.3 discusses the works that perform function analysis in order to uncover vulnerabilities as well as how to discover the relations between functions. Lastly, section 3.4 summarizes the remaining knowledge gap that this work resolves and which technologies are utilized to achieve this.

3.1. Static analysis techniques for deriving binary characteristics

We need to identify the base-address offset before we can analyze a binary firmware image. This is required as an incorrect offset will prevent absolute pointers and vector table entries from referencing the correct memory regions. This is illustrated in figure 3.1. This figure shows how variables that carry static memory references are influenced when analyzed under an incorrect base address offset. Several works have been concerned with developing static analysis techniques to uncover the potential base address and other information, such as its instruction set architecture.

De Nicalao et al. have proposed a way for code and data separation to ease the static analysis of binaries [26]. As perfect disassembly of machine instructions is undecidable [27], De Nicalao et al. propose a new technique for code and data separation based on sequential learning. More importantly, however, if the ISA is unknown, ELISA leverages a logistic regression model to identify the correct ISA and its endianness from the contents of a file. This, in turn, also enables the identification of the base address offset based on the identified ISA. This ISA identification is achieved by computing the possible byte value frequencies as they assume that compiled executables have different Byte Frequency Distributions (**BFDs**) when compiled for different CPUs. While this method of uncovering the base address is effective, it is not the main goal of the implementation. However, it proves that uncovering the base address is feasible without assuming partial system knowledge.

The work by Wen et al. provides an alternative method of base address identification in their work called FirmXRay [13]. The work focuses on detecting Bluetooth link layer vulnerabilities within bare-metal firmware. In order to achieve this, they introduce a novel method that allows for robust firmware disassembling despite the absence of system knowledge. Whereas other works (i.e., [28]) focus on the function prologues to enable base address identification, FirmXRay utilizes the fact that absolute pointers must point to certain instructions or variables with respect to their types. This insight enables using the present relation between a pointer and the variable or instruction it points to infer a potential base address. Absolute pointers are recognized by leveraging the functionality of load instructions and software development kit (**SDK**) functions that take strings as parameters. When all absolute pointers are identified, FirmXRay determines the most likely candidate for the base address by comparing the sums of correctly resolved absolute pointers. The main benefit of this method is that it is entirely static and does not utilize any form of supervised learning. Moreover, it can be applied to any binary

executable and is shown to be effective in a case study concerning hardware developed by Nordic and Texas Instruments.

In this work, we utilize the FirmXRay framework to enable the automated analysis of RTOS-based firmware. We will adopt the base address identification algorithm in our work as this allows us to perform analysis without relying on partial knowledge aside from the target architecture and microcontroller unit (**MCU**) manufacturer. The approach, as implemented, does not require any modification, and the proposed solution can be used as is.

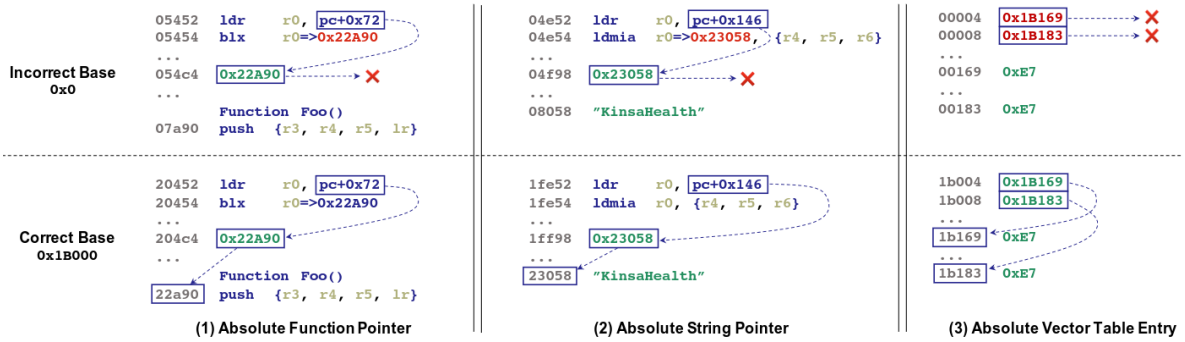

Figure 3.1: The effect of disassembling firmware with different base address offsets as provided by Wen et al. [13]

3.2. Function discovery

A function is nothing more than a deterministic transformation of input that produces an output based on that transformation. Functions can be deterministic but also stochastic, depending on the type of implementation that a programmer requires. One of the questions we need to address in this work is how we can identify functions without having any partial knowledge. This section provides an overview of different works that have concerned themselves with devising strategies for function discovery as well as the types of functions that can be found.

Sivakumaran et al. have provided a work to identify information about security-related configurations in stripped Cortex-M binaries called argXtract [15]. The goal of argXtract was to create a scalable solution that enables bulk extraction of security-relevant configuration data. To facilitate this extraction, argXtract performs four forms of identification: application code base identification, data identification, function block identification, and call of interest (**COI**) identification. The application code identification is performed by leveraging known address information of the core interrupt handlers, as these are stored within the vector table at specific offsets. They aim to identify the single control loop of the system in the default handler, meaning that this approach is only effective at identifying such base addresses in Type-III firmware that operates without the presence of RTOS libraries. The function identification proposed in this work relies on pattern matching and analyzing the call execution path. This means that while an approach for application code identification exists and can be utilized to gain insight into the device's security settings, it is limited to bare-metal firmware that does not utilize RTOS libraries. Furthermore, the work relies on partial knowledge of the system, such as the presence of a single control loop and the known address information of the core interrupt handlers.

The work by Gritti et al. called Heapster utilizes a different approach of function discovery to identify memory-related vulnerabilities [14]. In this work, Gritti et al. observe that allocator functions, which are critical for systems to perform their function, are often vulnerable and exploited. As memory allocators are extensively utilized in embedded devices, these devices require more scrutiny from the community, given their broad deployment in critical infrastructure. As such, they have created a solution named Heapster to identify heap management library (**HML**) components and subsequently test the security of its components. In order to identify functions that belong to the heap library, Gritti et al. simulate functions present in the binary to determine whether or not they are HML components [14]. First, all functions in the binary are analyzed and filtered based on numerical characteristics. This excludes functions that are too large or too complex. After filtering, the functions are simulated, and their implementation is compared to the expected behavior of HML functions. This was done by creating models that describe HML functions with predictable behavior. Modeling function behavior was done by cre-

ating sets of virtual input and output data where the output data showed transformations based on the expected function behavior. The input datasets are provided to these functions, and their functionality is verified based on whether or not the implementation matches the expected output results.

Heapster is focused on identifying HML vulnerabilities, but the technique used to identify HML components can be applied to any function with predictable behavior. While we cannot make any assumptions about the predictability of the behavior of functions that implement thread creation, we have observed a strong relation between thread creation and heap modification. While we cannot create models to identify functions that implement thread creation based on transformative operations on input data, we can identify heap-modifying functions using Heapster's approach. As such, by deploying other techniques, such as reaching definitions analysis, we can leverage the work provided in Heapster to identify the functions we need to find user application code in RTOS-based firmware.

3.3. Automated vulnerability discovery

This work focuses on identifying user-oriented application code in stripped RTOS-based firmware. Identifying user application code allows for targeted analysis to identify vulnerabilities and patch them accordingly.

Eschweiler et al. have created an implementation called discovRE that enables the identification of bugs through efficiently identifying similar functions within different systems [12]. This work implements an algorithm that can identify similar functions despite different optimization levels, compilers, operating systems, and CPU architectures. This is achieved by taking a known vulnerable function and comparing its structural and numerical features against the features of other functions to identify similarities. Examples of the numerical features that are taken into account are the number of instructions, number of basic blocks, and size of local variables. Examples of structural features that are considered are the control flow graph (**CFG**) of the function and other features of the basic blocks, such as the number of call targets. However, the work also leverages the presence of debugging symbols and function names, aspects we assume to be absent. As such, this approach is not expected to work in stripped RTOS-based firmware.

Gustafson et al. observe an increase in devices that are abandoned, reach their end-of-life (**EOL**) cycle, are no longer supported, or will no longer receive security updates [29]. Furthermore, they notice that devices that use monolithic firmware are especially vulnerable, given how these are difficult to examine. To combat this, they present Shimware, which is tasked with implementing new security measures for monolithic firmware images. This is achieved by identifying input-output (**IO**) pointers, empty memory regions, and self-referencing sections. These parameters allow Shimware to determine the type of vulnerability and to deploy an appropriate patch in a safe memory region while accounting for potential self-checks, which aim to prevent firmware modifications. The implementation of Shimware is limited to the analysis of firmware that is not based on an RTOS or RTOS libraries, as these pointers would be abstracted by the HAL. Furthermore, we note the usage of `angr` to perform variable recovery and state initialization and manipulation. They also use `angr` to generate the CFG upon which further analysis is performed.

KARONTE is a framework created by Redini et al. that enables the detection of insecure interactions within multi-binary systems [11]. The main goal of KARONTE is to improve the static analysis of non-standard configurations, as many devices implement their functionality through multiple binaries. In order to identify insecure interactions, Redini et al. utilize taint analysis to track the data between the different components. The relations between different components are identified by means of creating a binary dependency graph (**BDG**), which models the interaction between different binaries in the system.

In summary, the work by Redini et al. focuses on the identification of inter-binary relations in Linux-based firmware. The important takeaway from their work is that multi-binary relations, which are modeled through a binary dependency graph, can be leveraged to reduce false positives [11]. We utilize the same principle of leveraging strong relations between system components to reduce the amount of false positives we could encounter. While KARONTE mainly focuses on Linux-based firmware, we provide a framework for RTOS-based firmware that leverages the relations between specific functions and functional components to more accurately identify user application code. The work by Gustafson et al. focuses on the retrofitting of bare-metal firmware images in which no RTOS libraries are present [29]. They leverage the fact that IO interactions are not abstracted by a potential HAL to identify vulnera-

bilities. Like the Shimware implementation, we can utilize `angr` to generate the CFG and manipulate function states to recover variables, but we will not need to perform taint analysis. Lastly, we can leverage the numerical and structural features of stripped RTOS-based firmware to identify functional components and application code as performed by Eschweiler et al. [12]. However, the features we are interested in are different, and we will not base our identification on similarities between implementations as we assume no partial knowledge. However, we can leverage block structures and branching calls to specific functions to estimate the type of component we are inspecting.

3.4. Knowledge gap resolution

The previously mentioned works aim to tackle various challenges within the field of automated binary analysis, ranging from automated patching to code and data separation. However, many of these technologies are limited in their applicability as they target either bare-metal firmware without RTOS abstractions or Linux-based firmware. Furthermore, none of the existing works have investigated the possibility of leveraging and discovering RTOS library components to identify user applications on a system. We also find that several works perform automated patching by analyzing every function in a binary image. This means that functions that an attacker can never reach will be marked as vulnerable regardless. Leveraging RTOS components to identify application code in RTOS-based firmware will reduce the number of candidate functions that require inspecting, thereby optimizing the analysis process and reducing the number of unreachable candidate functions.

4

Approach

This chapter elaborates on the approach and methodology of this thesis. Section 4.1 provides a high-level overview that explains the phases of the approach, the required information, and why this information is necessary. Furthermore, it elaborates on the state of the system after each phase and provides context regarding the goal of each phase within the system. An in-depth description of each phase is given in section 4.2, section 4.2.3, section 4.3 and section 4.4 respectively. Finally, section 4.5 provides information on the challenges that must be addressed before PinDown can identify user application code.

4.1. Approach overview

In order to identify the different functional components, we need to perform three separate stages of analysis. Figure 4.1 provides a schematic overview of the pipeline. The goal of the first analysis stage is to establish basic information about the binary using state-of-the-art technology. This stage will be referred to as the *prerequisite analysis phase* (or the first stage of analysis) as the stage is concerned with uncovering basic information of the binary firmware image. This prerequisite analysis stage utilizes identification strategies described in FirmXRay [13] and Heapster [14] to do so. Furthermore, we will leverage the `angr`[1] framework to perform register and state manipulation, which allows us to identify basic heap modifying functions as well as perform *reaching definitions* (**RD**) analysis. Assuming the hardware manufacturers Nordic and Texas Instruments and CPU architecture Cortex-M, FirmXRay enables us to recover basic information about the system, such as its base address and entry point.

After the FirmXRay implementation has finished its analysis, the base address and binary entry points are stored in a `conf.yaml` file that can be parsed by the next component of the prerequisite analysis phase called Heapster[14]. We use Heapster to identify basic functions that are part of the *heap management library* (**HML**) and pointer source generators as these enable the identification of other important functions. As such, we only need to execute the first two steps of the Heapster framework.

The result of executing FirmXRay and Heapster is that we now have knowledge of the base address, the entry point of the binary, the addresses of essential functions that modify the heap (such as `memcpy`, `memset`, `memcmp`, `strlcpy` etc.) and a list of function addresses that generate pointers. Given this information, we can initiate the process of discovering *caller candidates* by applying RD analysis to each identified heap modifying function. We define a *caller candidate* as a function that calls any of the essential heap functions recovered by Heapster.

Real-time operating systems have a need to manage their processes by means of handling threads. The main insight we have is that amongst these caller candidates, there is a function tasked with modifying space on the heap for managing the application present in the binary image, as heap modification is required to accommodate user applications.

The second stage is called the *functional component identification phase*, which will scan through the call graph with the entry-point function as the root node. We will use the `angr` framework to manipulate registers, identify function relations, and uncover function relations by analyzing the control flow

[1] `angr` framework: `https://angr.io/`

graph. In this scan, we will identify the usage of the previously identified caller candidates. If such a function is used, we will heuristically determine whether this function creates threads in order to launch user applications. The end result of this analysis stage is a list of functions that either directly launches user applications or initializes a functional component.

The analysis of several RTOSs shows that the function that is passed to such a caller candidate as an argument is likely to play a critical role in the initialization of application code. As such, the third stage is called the *application code discovery phase*, which is tasked with discerning whether we are dealing with application code (*option* ①) or with a functional component that itself is tasked with initializing application code (*option* ②). This means that the address is either the application code, which we can identify with heuristics, or we can derive that the address is a helper function that launches the application. If we encounter option ①, we can utilize several characteristic filters to determine whether or not we are dealing with application code. In the case of option ②, we can leverage the presence of caller candidates to identify the application code. Since the *functional component identification phase* has already identified likely caller candidates based on the fact that they require a valid function as an argument and are called in the upper layers of system initialization, we have narrowed the list of potential caller candidates considerably. Assuming that only one function is tasked with creating threads (as duplicate functions indicate poor implementation standards), we search the upper layers of the call graph where the function passed as an argument is now appointed as the root node. When we find a function that matches the caller candidates identified in the *functional component identification phase*, we again retrieve the function arguments that resolve to valid function addresses. Our analysis of RTOSs shows that these valid function addresses are the functions that contain application code, and we label them as such.

Finally, we are left with a list of function addresses labeled as application code, functions that are functional component candidates, and an inconclusive list of functions that could be a functional component or application code and further inspection of these candidates is required. The inconclusive candidates were taken into account as discerning between applications and supporting framework functions is not feasible using an automated process relying on static analysis.

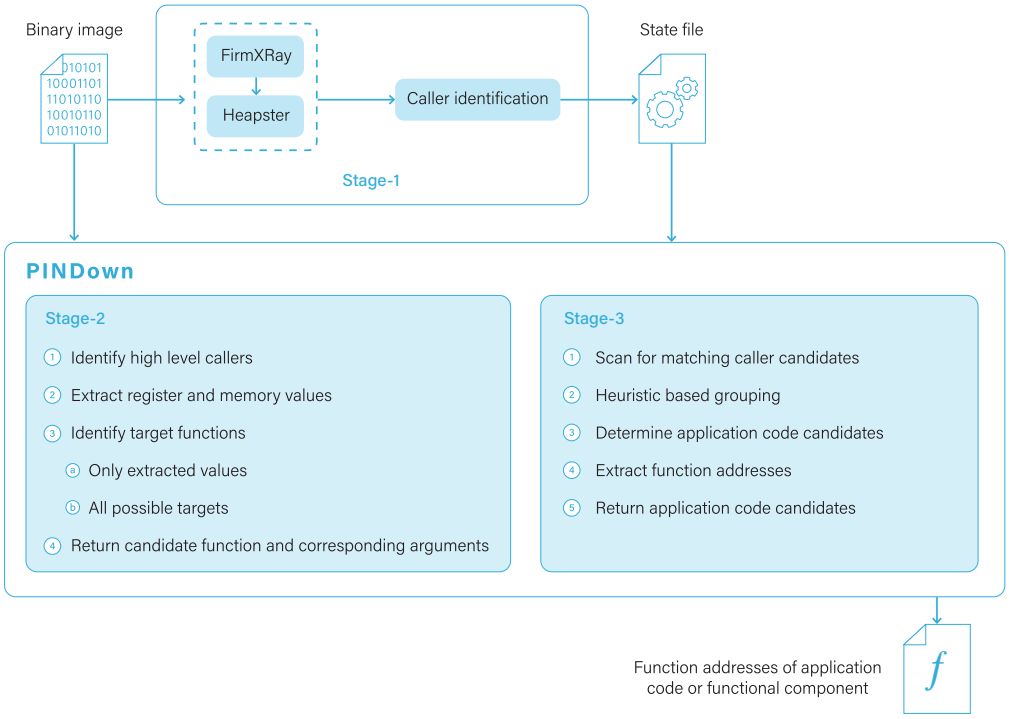

Figure 4.1: A schematic overview of the relations between components in the framework and the steps that are taken to produce intermediary results. A binary firmware image is provided and is analyzed by FirmXRay. The base address and binary entry point are then passed to Heapster. Caller candidates will be identified based on those results and the function addresses will be saved. These addresses are then used by PinDown in two separate stages in order to identify user applications within that binary firmware image.

4.2. Prerequisite analysis

We have identified the crucial components that we need to identify based on the results of our analysis on RTOSs, and these components are present in all operating systems that we have chosen to inspect. We want to construct an approach that can statically analyze all binary firmware images for all these RTOSs and determine to what degree this process is generalizable.

In order to keep our approach as generalized as possible, we need to identify these components in a way that does not depend on specific characteristics related to RTOSs. We must inspect functions based on their behavior and role within a system. Each binary firmware image has a base address and binary entry point, and each RTOS from our selection utilizes thread creation to initialize user applications. We can leverage state-of-the-art technology to identify both the base address and entry point address and the functions that enable the discovery of such a function that creates threads, namely heap modifying functions.

After we have been able to identify the base address, binary entry point, and heap modifying functions, we need to identify possible candidate functions that may create threads. Given how thread creation generally involves the modification and allocation of space on the heap, we can leverage knowledge of the previously identified heap modifying functions to generate a list of caller candidates. This can be achieved by employing static analysis techniques.

We call the first stage of analysis the prerequisite analysis phase. This analysis stage focuses on utilizing several techniques to identify the addresses of caller candidates. The result of the prerequisite analysis phase is a list of function addresses that PinDown can later use to determine the addresses of functional components and user applications.

4.2.1. Base address identification

The base address offset needs to be correctly identified. An incorrect offset yields the incorrect disassembly of pointers that contain memory addresses of functions. If the base address is incorrect, the address to which a pointer points may have been disassembled incorrectly and not contain the function we are interested in. In order to correctly identify the base address, we employ a state-of-the-art identification technique called FirmXRay[13]. As explained in-depth in chapter 3, FirmXRay uses *robust firmware disassembling* to identify the base address of a given binary. This result is achieved by realizing that absolute pointers will point to certain variables or instructions depending on the variable type or instruction. As we require the proper disassembling of functions for our approach to be effective, we implement this strategy of base-address identification in our framework. The implementation of FirmXRay is focused on performing the analysis within hardware developed by either Nordic or Texas Instruments. As such, we need to take this constraint into account when constructing and testing our approach on binary firmware images.

4.2.2. Basic heap modifying function identification

We need to be able to identify *basic functions* that are part of the HML present on the system before we can identify functions that create threads. Our analysis of the chosen operating systems has shown that functions that create threads depend on functions in the HML that perform heap modifications. In order to identify these functions, we can leverage a state-of-the-art technology called Heapster[14]. Heapster identifies functions based on predictable and expected behavior in conjunction with filtering based on generic function features such as size, the presence of loops, and other characteristics. Identifying functions based on predictable behavior is done by modeling data within memory regions and how this data is transformed by a function. As these basic functions are expected to perform specific transformations, Heapster checks if the transformation matches a specific function model. Heapster has provided several models that describe the expected behavior of functions that are of interest. Heapster then tests each function in the firmware image against these models to see if there is a match. In total, there are four models that we can utilize in order to find functions of importance given their perceived role in thread creation, namely `memset`, `memcmp`, `memcpy` and `strnpy`. We will leverage this strategy to discover these functions in a firmware image.

In order to perform this analysis, we provide Heapster with the base address and binary entry point as discovered by FirmXRay. After Heapster is finished simulating the selected functions, all results will be put in a `hb_state.json`. This file will be utilized and modified by subsequent steps in the Heapster framework.

4.2.3. Caller candidates discovery

The next step in the first stage of analysis is to discover *caller candidates* given a list of *basic heap modifying functions* and potential *pointer source generators* as a result of executing FirmXRay and Heapster. We define *caller candidates* as functions which, at some point in the call graph, invoke *basic heap modifying functions*. Due to how functions that create threads are usually implemented, these functions will not be discovered by the *pointer source generator identification* that Heapster performs as they typically do not match the criteria of the functions intended to be identified by Heapster. However, the approach implemented in Heapster can be leveraged to enable the discovery of caller candidates by logging every function that, at some point, invokes a heap modifying function. This allows for a broader subset of functions to be identified, upon which we can then perform analysis.

The end result of this discovery phase is a group of memory addresses that will contain the address of a function responsible for thread creation, which we can later narrow down by leveraging other characteristics. This result is guaranteed based on the analysis we performed on such functions in the chosen RTOSs.

In order to identify caller candidates, we adjust the implemented strategy of Heapster[14] and employ *reaching definitions analysis*[24] (**RD**). Given the addresses of *basic heap modifying functions*, we need to identify every call-site in every function in the binary that invokes such a function. A *definition graph* will be generated at the perceived call sites that target heap modifying functions. A definition graph is a directed graph where nodes are represented as register definitions and edges represent definition dependencies [14][24]. We are interested in the nodes that represent function arguments as these indicate definitions provided by caller candidates. When such a node is found, we utilize the strategy presented in the work by Gritti et al.[14] and build corresponding definition graphs, but now starting at the function where this type of node is found. This process is repeated until we no longer find a node that can represent an argument provided by a potential caller candidate. The function that was reached when the process of backward analysis terminates will be labeled a caller candidate.

The result of this stage of analysis is a list of *caller candidates*, which have been reached by performing an RD analysis on all basic heap modifying functions that were found by executing FirmXRay and Heapster.

4.3. Functional components discovery

We have been able to establish basic information about the binary we aim to analyze by applying the approach in section 4.2 and have uncovered functions that are of particular interest by using the approach from section 4.2.3 as amongst the acquired functions will be a candidate that is tasked with the creation of threads. This has been determined to be the case based on the inherent functionality these functions must provide to fulfill their purpose. Given how we have established some properties of the caller candidate functions discovered during the prerequisite analysis phase, we can now start applying characteristic filtering. We now begin the second stage of analysis, which we call the *functional component discovery* phase. In this analysis stage, we will use the results from the previous stage, which were stored in a state file.

The end-result of the second stage of analysis is a list of function pairs $\{(f_1, f_2) : f_1 \in \mathbf{F}_1, f_2 \in \mathbf{F}_2\}$. In this representation, $\mathbf{F}_1$ is defined as a subset of caller candidates found in memory regions associated with system and user application initialization. Functions from the set $\mathbf{F}_2$ were found to be passed as arguments to functions within the set $\mathbf{F}_1$. This means that the set $\mathbf{F}_2$ exists out of valid function addresses found at call sites that invoke functions from the set $\mathbf{F}_1$.

4.3.1. Call-graph analysis

The initialization of application code in the devices defined in our scope in section 2.5 usually happens in one of the first few layers of the call graph, as has been further corroborated by our analysis of the selected RTOSs in section 2.3. Based on the results of this analysis, we limit our function discovery to the first three layers of the call graph within a binary firmware image where the function at the entry point of the image is determined to be the root node. Starting at this function f_0, we will check whether the call targets of corresponding call sites match one of the functions in the resulting list of *caller candidates* at the end of the first analysis stage. This process of finding matching functions will be repeated for each function of the call graph until a depth of three has been reached, meaning that we will enter each encountered function as long as there is no match found within our list of *caller candidates*. The

matching address and location will be stored if a matching function is found. The result of this step is a subset of caller candidate functions, which are initialized in the first few layers of the call graph, as analysis has shown that thread creation to execute application code takes place soon after the `reset_handler` of a device is called.

4.3.2. Register and load instruction retrieval

Given a list of localized caller candidates resulting from the previous step, we now need to inspect the arguments with which this function is called. We inspect each instruction of the block of each respective call site with a target function present within our subset of caller candidates. As the calling convention for ARM indicates that registers `r0` through `r4` typically contain function arguments, we perform symbolic execution to retrieve the values within these registers after the function is called. In order to acquire this information, we initialize a blank state at the start of this function and step through each block until we have reached the block that contains the call target we are looking for. After stepping through one more time, each register contains information with which the function has been called. We need to expand our instruction analysis since we cannot make assumptions about the amount of arguments such functions may have or which registers contain arguments that we are interested in. Function arguments may, for example, be stored on the stack at some offset from the *stack pointer* (`sp`), which we cannot reach by means of symbolic execution. In order to acquire such arguments, we need to inspect the values loaded from memory in each block of instructions containing a caller candidate as a target. The instruction in these blocks whose contents we need to analyze is the `ldr` instruction. This instruction loads values from memory into a specific register. This operation is performed by taking the value of the *program counter* (`pc`) to which a static value is added. The final result of this addition is a memory address at which the information is stored that needs to be loaded into a register. This form of addressing is also known as *program counter relative addressing*[2]. The symbolic execution in ANGR will perform these operations automatically, meaning that whatever value is loaded into a register can be overwritten in the same block. We can overcome this by explicitly calculating and retrieving the values of the program counter and the value that needs to be added to the `pc` to uncover the memory location from which a value is loaded. We then store all the values found within registers and all values loaded from memory using `ldr` instructions in a list.

At the end of this step, we should now have a list of function arguments, their precise values, and any loaded values from memory for each *caller candidate* present in the first three layers of the call graph.

4.3.3. Valid functions

Given how functions that create threads pass functions as arguments to execute them, we can narrow down the list of *caller candidates* further by determining whether or not one of the arguments that belong to each respective function represents a valid function address. Using `angr`, we generate a *control-flow graph* (**CFG**) and call graph for the binary we are inspecting. These graphs will be populated with every function that has been identified during `angr`'s CFG analysis, and `angr` stores these function addresses as well as other values in its *knowledge base* (**KB**). In order to determine whether an argument value is a valid function address, we only need to check if this value is present in the knowledge base of the CFG analysis. Our analysis of RTOSs has shown that user application code may be stored and invoked through several means. Such function addresses may be passed directly to a caller candidate, meaning that the address that is passed is the address of the function that hosts application code. Arguments may also be presented as pointers, where the pointer points to an address in memory that, in turn, points to application code. Lastly, we found that several RTOSs contain data structures that resemble lists and that these data structures can contain addresses of several user applications. In RTOSs that utilize data structures to store application code addresses, we found that a pointer to the head of the list may be passed as an argument in conjunction with a value that specifies the index. When analyzing function arguments, we need to check whether the argument value points to a valid function address when interpreting that value in line with either of these three scenarios.

After this step is completed, we should have narrowed down our initial subset of caller candidates to only those candidates that pass valid function addresses as arguments with the intent of modifying the heap to accommodate these functions and executing them. We label these functions as *TC*

[2]ARM Documentation on pc relative addressing:`https://developer.arm.com/documentation/dui0473/h/Cacdbfji`

candidates and store the respective valid function addresses with which they were invoked and label these as potential *functional component candidate* or *application code candidate*. We define a *functional component* as a function that performs some other system initialization before invoking a user application, and application code is a function developed on top of the real-time operating system to perform a task as intended by the creator of the device.

At the end of this step, we will have a list of valid function addresses that were passed as arguments to functions from our subset of caller candidates. We also have a modified and reduced list of caller candidates, which represents a subset of caller candidates resulting from the first analysis stage.

4.4. Application code discovery

The third analysis stage determines the addresses of user application code based on the acquired set of TC candidate functions. Our previous analysis phase has provided us with a list of *TC candidate* functions and the corresponding function addresses that were passed as arguments, from here on out, referred to as *sub-function* (**SF**).

Assuming that one of our TC candidates is a function that creates threads, we need to investigate whether the corresponding function passed as an argument hosts application code or if the function plays a part in initializing the application of the firmware as a functional component. Unfortunately, we cannot make any assumptions about the construction, the utilized libraries, or the tasks of application code, as these components vary between RTOSs and devices. This makes the process of differentiating between what components classify as application and operating system-related components challenging. However, we have isolated a set of operating system-specific functions, which are provided with function addresses as arguments during the second stage of analysis. These functions are assumed to be operating system-specific and will likely not be invoked by applications written for RTOS-based hardware. This is because the RTOS provides resource management for the developer, which is one of the main reasons for employing an RTOS. As TC candidate functions were called during the initialization of the entire system, it is unlikely that an operating system would provide these functions to whatever application runs on top. The usage of such functions is sparse and strongly associated with the initialization and calling of application code. This implies that the regions in which such functions are found should be analyzed for the presence of application code or a functional component. Using the results of the RTOS analysis, we can perform a superficial analysis of the acquired sub-functions to determine whether we are analyzing a functional component or a user application. In this analysis, we will reduce the number of candidate functions likely to be application code by leveraging features identified during RTOS analysis. Such features include the nested presence of the TC function to which the current function was passed as an argument, the amount of `bl` instructions, whether their respective function blocks contain multiple instructions and the number of predecessors and cross-references belonging to a function. However, it is possible that the features we will look for will not be present in the component that we are analyzing. In this case, we can still attempt to make some observations based on our analysis of operating systems; however, the results will be labeled *inconclusive* instead.

The final results of the third stage of analysis is a group of functions that are likely to be user applications. This phase will also separate the functional component from the application code based on the heuristics, as mentioned earlier. If no functional component is involved (i.e., the application code is initialized directly by passing its function to a TC-candidate), we will perform a superficial analysis to determine this.

4.4.1. Component grouping

We start the analysis of the sub-functions we have identified at the end of the second stage of analysis. Our analysis of RTOSs shows that there are a couple of possible compositions within the chosen group of frameworks. To reiterate, these groups are as follows:

1. The sub-function utilizes thread creation to initiate some auxiliary attributes before application code is invoked
2. The sub-function utilizes thread creation to invoke the actual application code
3. The sub-function makes no usage of thread creation and makes some static function calls to initialization functions before application code is invoked
4. The sub-function is the actual application that runs on top of the system, and no further initialization is required

We first need to analyze the discovered sub-functions associated with the TC candidate in order to determine which classification of the component we are dealing with. Our RTOS analysis has shown that the initialization of user applications in functional components happens in a superficial layer of the call graph starting at the root of the sub-function. Based on this analysis, we have determined that it is sufficient to search through the first two layers of a call graph where the discovered sub-function is the root node.

During this scan, we are looking for functions also present in our list of TC-candidates, which was the result of our second analysis stage. First, we search for the TC-candidate that invoked the current sub-function. If we locate this same function in this sub-function, it is more likely that our TC-candidate will indeed create threads. We base this on the insight we gained in our RTOS analysis, which shows that these functions will be used both for initializing a functional component and within the functional component. Establishing the presence of such a function indicates whether we might be dealing with either a class that depends on thread creation to initialize user applications (class 1 or 2) or class 3 or 4, which do not depend on thread creation. If another TC-candidate is found in any of the scanned layers of the sub-function, we perform register and load instruction retrieval as we did during the second analysis stage. This will provide us with the associated values loaded from memory and argument values found in registers. Much like in the previous phase, we will again scan these values to see if valid functions were passed as arguments. Such a function may have been compiled as part of the system, but no functions have been passed that require initialization. In such an event, these registers and loaded memory values will not yield any valid functions. If a valid function address is found, then we mark this function as an *application code candidate* as this function needs to run on top of the previously initialized system and thus likely fulfills the intended use of the system. If no values are loaded into relevant registers or if the registers do not contain a valid function address, then assume that no application code is initialized by means of thread creation, and instead, there will be a function call in a block with only a single `bl` instruction. RTOS analysis has shown that if no functions are initialized by means of executing threads, then there is a single call to a main function that contains application code. It is possible that the function value is not within the list of functions present in the binary, so we have to check the validity of the discovered address.

When we do not find any functions that match the sub-function its associated TC-candidate and when no function matches *any* of the functions in our list of TC-candidates, we assume that we are dealing with either class 3 or 4. As we cannot make any assumptions about the internal workings of applications or the frameworks that launch them, there exists the possibility that the current sub-function we are inspecting is a user application or that we are inspecting a component that launches application code. In either case, manual verification is required of these addresses as we lack any context about the system as a whole. As such, while we can apply a superficial analysis to find probable user application code, we have to mark the component that we are inspecting and the functions we find based on this analysis as inconclusive candidates. This will increase the false positive rate, but there is still a very likely chance that we have identified the components in which we are interested.

The result of this phase of analysis is a function or group of functions that are labeled as user applications, functional components (if one was present), and inconclusive function addresses that we suspect to be application code but cannot verify this in an automated manner. In the case of inconclusive results, we must manually analyze the provided functions to correctly identify the actual application code. The manual analysis of individual functions should be relatively simple but requires at least the knowledge of the device's applicability to which the firmware belongs. Lacking knowledge or context about the device for this analysis will make it hard to accurately assess whether the identified functions perform the intended functionality of the device. Furthermore, the evaluation of PinDown has shown that in systems with only a single user application, PinDown generally identifies less than seven candidate functions. This is a reasonable amount of functions to analyze compared to the several hundred that require inspection without the usage of PinDown. This step concludes the PinDown analysis.

4.5. Addressing improper firmware decompositions

Due to the nature of firmware disassembly and the issues that THUMB-2 instructions introduce because of the varying instruction sizes, it is possible that state-of-the-art technology can struggle with correctly handling firmware. Examples of such handling issues are misidentified function entry points, incorrectly

interpreted firmware instructions and operations, or the incorrect interpretation of data regions. These issues enable the possibility that functions may not be accurately disassembled. This is problematic for our analysis approach as we depend on finding certain functions to identify critical components. As such, we need to account for these problems as best as possible to enable firmware analysis. In this section, we explain the problems that we encountered during firmware analysis and how these issues are partially solved.

4.5.1. Functions disassembled as callsites

During `angr`'s function analysis, it is possible that the target function of a call site is not interpreted as a separate function but instead as a continuation of that function. This is a function identification issue. The field of function identification is well studied, and new algorithms are being devised to reduce overhead and increase identification accuracy [30][31]. The result of calls to incorrectly identified functions is that a branch is created into a distant memory region. Furthermore, we find that the original function now contains the instructions of two functions that should have remained separate. Not only does this cause the original function to *absorb* the call-target, but it also makes the function address of the absorbed function unidentifiable. If this occurs within one of the functions that we require during analysis, then we will not be able to complete the analysis in an accurate manner. Analysis has shown that this issue tends to occur at the last call site of a function. As this problem is related to the `angr` framework, which we use to perform automated analysis, identifying the root cause and implementing a fix is outside of the scope of this research. However, we can introduce a workaround that attempts to identify the originally intended function at the cost of potentially introducing a false positive result during analysis.

All disassembled function addresses are stored in a knowledge base. As the intended function address is not identified as a function during disassembly, this address will be missing. However, it is possible that the disassembler will identify the corresponding function prologue. As the disassembler identifies this region as a part of another function, it will attempt to make sense of the situation by looking at nearby memory addresses and providing these as the corresponding function addresses despite this not being the actual case. This behavior has been observed during the analysis of several binaries. The result was that the call target instead had an address value with a possible offset of a few bytes added or subtracted. While this replacement function may not be identical to what the function was intended to look like, it most likely still contains the instructions that are relevant for continuing analysis.

There are two ways this issue can manifest itself. In the first case, the function whose call-targets are being analyzed branches off into a memory address `before` the start of the current function. In this case, we can be sure that the call site that branches off into a distant memory region was not correctly identified as a function, as it points to a region before the address of the current function. For example, if we find a function at address `0x1000` but it has a call-site at address `0x200`, then we know that the call-target at the corresponding branching instruction was instead interpreted as a call-site, which in turn results in this corresponding function address not being directly accessible. In the second case, a call site branches off into a memory region far `beyond` the actual end of the function we were inspecting. In this case, we can verify the presence of the issue and see if there is a gap between the address of the last function call at which the branching was identified and the memory address to which it points. More problems arise in the second case, as it could be that a branching call points to the memory region directly after the function we are analyzing. This means we need to estimate a function's size (amount of instructions), which is no longer reasonable. To remedy this misidentification, the best we can do is to search for valid function addresses in our knowledge base that are in numerical range of the target instruction addresses identified at the incorrectly interpreted call site. For example, if an incorrect call-site has been identified at a branching instruction that points to address `0x164`, then we search our knowledge base for neighboring function addresses (such as `0x160` and `0x170`). We then check if this function contains an identical set of instructions found at the incorrect call site. If this is the case, we store this function and proceed with the analysis with this function instead. We will attempt to identify any valid call targets if no such function was found. However, it is highly likely that the call site will not contain all the instructions of the original function, and as such, we may not be able to identify any other call targets. In this case, we halt analysis.

4.5.2. Unreachable critical functions

During the first stage of analysis, the backward generation of definition graphs may terminate because of the misidentification of functions. When this process of analysis terminates prematurely, it means that we have not reached the caller candidate that we were interested in. This issue then propagates through the following analysis stages as we depend on the generation of a set of candidate functions in order to identify functional components and user applications. This is a hard problem to solve as we cannot discern between correct termination and incorrect termination.

In order to still attempt analysis despite the issue of incorrect caller candidate generation, we can employ a brute-force approach to enable further analysis in the second stage of analysis. Instead of scanning for matching functions from the list of acquired caller candidates as a result of the first stage of analysis, we can instead inspect *any* function call where registers and loaded memory values contain valid function addresses during the second stage of analysis. The main drawback of this method is the potentially larger set of TC-candidate functions that we acquire. This means that the analysis will take longer to complete. Furthermore, it is possible that we incorrectly identify more components as potential application code candidates. Nonetheless, using this method in our approach opens up another avenue for identifying user applications despite the lack of caller candidate generation. However, this comes at the cost of acquiring a higher false positive rate and thus requiring more extensive manual verification.

4.5.3. Function abstraction in optimized firmware

When a firmware image is compiled with an optimization setting, the goal is to optimize the functionality of the resulting binary. This means that some functions may be *abstracted* away or `inlined`. The final result is that the binary contains fewer functions than the code that was compiled and that some functions no longer have a 1-to-1 equivalent. This process of inlining functions makes manual analysis difficult if one tries to compare functions in source code to functions in binary disassemblies. Due to how we identify TC-candidates by inspecting the layers of a generated call graph, function inlining poses a problem for our approach. This is because optimized binaries are structurally different from unoptimized binaries. We need to construct a generalized approach that could be applied to any binary that fits our scope.

In order to address this issue, we can implement a workaround that identifies *transitive components*. We define a *transitive component* as a component in the call graph that only has a single predecessor and could have multiple successors if there are no cycles within the function. If we encounter such functions during analysis, we abstract away this *transitive component* ourselves, effectively reducing the depth of the current call-graph level by one. During the operating system analysis, we found that several wrapper functions are employed that do not seem to perform any critical functionality other than providing clarity to developers. We need to extend our approach to account for this issue, as the presence of such functions in unoptimized firmware can prevent us from reaching the critical components we are interested in, as we have based the required depth of analysis on uncovered characteristics. To illustrate this, if function $\mathbf{F}_A$ calls a function $\mathbf{F}_B$, function $\mathbf{F}_B$ only calls a single function $\mathbf{F}_C$ and function $\mathbf{F}_B$ is only invoked by $\mathbf{F}_A$ then we abstract away $\mathbf{F}_B$ by stating that $\mathbf{F}_A$ calls $\mathbf{F}_C$ by means of a *transitive component* and continue analysis, ignoring the additional layer introduced by $\mathbf{F}_B$. The benefit of this approach is that the entire generated call graph will remain the same while allowing us to store intermediate results and ignore layers that are introduced by functions that serve no other purpose than to call other functions. Using this approach, we can find functions that are of particular interest in both optimized and unoptimized binary firmware images.

4.5.4. Infinite looping during symbolic execution

It is possible to encounter indefinitely looping structures when using symbolic execution to retrieve values stored in registers before a function is invoked. Being able to determine whether or not such a loop terminates is a problem that is reducible to the *halting problem*, which has shown to be undecidable [32]. While there exist techniques that leverage symbolic execution to identify infinite loops, such as the work by Ibing et al., this detection was not present in the `angr` framework that we utilize for analysis. Instead, we have implemented a more superficial way of exiting such loops.

During the symbolic execution of a function, we do not enter new function calls to prevent path explosion. It is possible that looping structures depend on the return values of such function calls and that without these values, such a loop will most likely never terminate. In this instance, there is no other option but to terminate the symbolic execution of the function in question, as the contents of register

values need to be kept track of throughout the execution of instructions within the function. This means that we will not be able to determine the contents of the registers before a function call is made, as register values are constantly shifted and changed. Unfortunately, there is no way to break out of such a loop while maintaining correct register states. In order to continue binary analysis in the event we encounter such a loop during symbolic execution, the best approach is to keep track of the number of loops and terminate execution if the number of loops becomes unreasonably large. As there is no proper definition of ”unreasonably large,” we have decided that if a loop is repeated more than 100 times, we terminate symbolic execution and mark the function as containing a loop that does not finish. Despite not being able to read register values, we can still inspect the `ldr` instructions of the function if values from memory are loaded into registers. When a non-terminating loop is encountered, this becomes the only way to identify potential function addresses that are passed as arguments.

4.5.5. Incorrect pc-relative offset

A side effect of incorrect function identification is that `ldr` instructions that depend on pc-relative addressing will no longer load data from the correct memory address as the program counter no longer holds the correct value that is required. These load instructions take the address value of the current instruction and increment it with a static value to determine the memory address from which to load data. In the event a function is instead identified as a call-site, that means the corresponding program counter will contain a value that will be off by at most two as the length of a `bl` instruction is two bytes. However, It is possible that the program counter’s value remains unaffected and that the proper value can still be loaded. This is another limitation of the technology we utilize, and we are not certain of the root cause of this behavior. However, the possibility that we cannot identify the information loaded from memory due to this incorrect pc-relative addressing still needs to be accounted for.

In order to do so, we can manually keep track of the program counter during automated analysis of the instructions within a function. As the generation of the control-flow graph also includes analyzing the presence of such load instructions and the target addresses from which a value is loaded, we can check if the instruction address that performs the load instruction is in the knowledge base of the control-flow graph. When this is not the case, we can utilize our custom program counter to retrieve the values from the target memory location. We will use our custom program counter to generate two different target values, one value with an offset of one from the instruction address and one value with an offset of three from the instruction address. As we are only concerned with potential function addresses, we only need to check if either of the retrieved values from memory is present in the knowledge base of the call graph. When such a value is present, we store this value and use it in subsequent steps of analysis.

5 Evaluation

This chapter provides context for the experimental settings and the generated datasets, based on which factors we determined the analysis to be successful, and argues why the results of the experimental validation show that our approach is effective in discovering application code. Section 5.1 discusses the details of the hardware that was utilized to perform the experimental validation and how the experiments were conducted. Section 5.2 provides information about our ground truth dataset, how this dataset was established, and what factors were considered when creating the ground truth. In the same section, we will discuss the results of each analysis phase and analyze them in order to verify the outcomes. Section 5.3 discusses the analysis results of the wild firmware dataset and how this dataset was created. The experimental results from applying PinDown to wild firmware provide insight into the applicability of our method on firmware built on top of other RTOSs and the RTOSs on which the approach was based. Lastly, section 5.4 will discuss the performance results and elaborate on how the performance results help explain the experimental validation.

5.1. Experimental Settings

All experiments were performed on consumer hardware using open-source software. A PC was outfitted with the Kali Linux rolling operating system, version 2024.2, using an x64 Threadripper 2950X CPU with 16 cores and 32 threads and 32 gigabytes of DDR4 DRAM. We decided to use Docker[1] to ensure the reproducibility of the experiments and results, as it allows us to create a containerized environment in which we have full control over software versioning. This prevents dependency issues from arising and shows that the results will be consistent regardless of the system on which the experiments are performed. Docker loads all requirements from the Dockerfile in the repository and installs all system requirements for the Heapster and FirmXRay frameworks, on which our implementation is partially dependent. The version of Ghidra with which all firmware was manually analyzed is also installed in this environment. From this version of Ghidra, a `.jar` file is generated, which will be used by FirmXRay. This was done to prevent features in newer Ghidra revisions from causing potential errors.

Several scripts were written in Python to perform the experiments. However, three main scripts perform the individual analysis stages:

1. Performing the prerequisite analysis on the *ground truth* dataset
2. Performing the prerequisite analysis on the *wild firmware* dataset
3. Performing the second and third stages of analysis for either firmware from the *ground truth* or *wild firmware* dataset.

Automating the serialized analysis of each single firmware image takes too long, given the number of available samples. The scripts that are tasked with executing the prerequisite analysis phase leverage multiprocessing. This enables parallelized analysis of all samples. While the hardware on which the experiments are performed does not provide enough computational resources to perform a complete parallelized analysis of all firmware samples, we can now analyze all images in the *ground truth* data set within 1.5 hours. This is considerably faster given how serialized analysis of each individual process would take ~ 8.5 hours. Due to technical limitations, FirmXRay analysis has to remain serialized. In

[1] Docker: `https://www.docker.com/`

order to avoid excessive waiting time, intermediary results provided by FirmXRay and Heapster in the prerequisite analysis phase are stored and used in later analysis phases. This means that we only execute FirmXRay and Heapster once per firmware image. The processes and analysis results of these state-of-the-art technologies are deterministic, and the input does not change between analyses.

The intermediary results of PinDown are comprised of several groups of memory addresses at which important functions may be hosted. In order to verify the correctness of our framework, the firmware that was automatically analyzed with PinDown is now manually analyzed and reverse-engineered. The application code is identified within each sample utilizing the results and guidelines from the RTOS analysis in chapter 2 if one is present. We then verify that the addresses found at the end of each phase match the components we intended to identify. We claim that stage one analysis is successful when the group of identified caller candidates contains the address of a function that creates threads. We claim that stage two analysis is successful if we have identified the caller that creates threads and that the arguments that belong to the corresponding function call contain valid functions that play a role in initializing application code. Lastly, we claim that stage three analysis is successful if the addresses that contain application code are included in the group of identified functions. Furthermore, for each analysis phase, we include a false positive rate to indicate how precise the identification is and the percentage with which we were able to reduce the number of candidate functions within a binary image.

5.2. Experimentation: Ground truth

The experiments on the ground truth dataset were performed with the experimental setup described in the previous section. We sequentially apply each analysis stage to each binary firmware image in the dataset and inspect the result of each stage. We have reverse-engineered each sample in the ground truth dataset and manually identified each function we aim to identify using PinDown. We will inspect the results of each analysis stage and verify that we have identified the intended function(s) at the end of each stage. This section provides information on our dataset, how each image was generated, and the results of each analysis stage.

5.2.1. Ground truth dataset description

We compiled a dataset of 24 firmware images at varying optimization levels, hosting different applications for varying development boards in order to establish a ground truth and to perform our experimental validation. In order to create the dataset, we have installed toolchains available on GitHub for all RTOSs that were selected; RIOT[2], mBed OS[3], Zephyr[4], Contiki-NG[5], NuttX[6] and LittleKernel[7]. As we have limited our scope to hardware that runs on Cortex-M and because our base address identification process relies on characteristics in micro-controller units (MCU), we need to select hardware profiles that match the requirements of our scope. The installed toolchains come with prebuilt modules and extensions that accommodate a large variety of different development boards. The libraries in these toolchains also provide several configurations, modules, and libraries that allow example projects to utilize unique hardware features.

For each RTOS, we aspired to compile two firmware images for both Nordic-based hardware and Texas Instruments-based hardware. However, some of the toolchains did not offer hardware profiles for both manufacturers. When we are not able to generate firmware images for both manufacturers, we focus on the hardware profiles that we *can* compile. These images are compiled with an equal distribution of enabled and disabled optimization flags. This means we have about an equal number of images compiled with O0 and O5; due to the limited availability of hardware targets that matched the aforementioned requirements, we have chosen as many varying development boards as possible but have resorted to previously chosen targets if necessary. While this could influence the eventual structuring of the binary at compilation, it is unknown to what degree this will be noticeable or whether it will affect analysis. As different development boards host different revisions of Cortex-M, we aim to exclude the possibility of biased results by applying analysis to as many different revisions as avail-

[2]RIOT OS Github: `https://github.com/RIOT-OS/RIOT`
[3]MbedOS Github: `https://github.com/ARMmbed/mbed-os`
[4]ZephyrOS Github: `https://github.com/zephyrproject-rtos/zephyr`
[5]Contiki Github: `https://github.com/contiki-ng/contiki-ng`
[6]NuttX Github: `https://github.com/apache/nuttx`
[7]LittleKernel Github: `https://github.com/littlekernel/lk`

able.

We need to decide what type of application the RTOS should be compiled with. We decided to compile each image with a different example application provided by the installed toolchains. This was done to exclude the possibility that incorrectly written custom applications would affect the results. Many examples of applications that are provided aim to show off certain properties of the associated hardware. We want to focus on varying degrees of software complexity instead of functionality. This means that we want firmware with only a single application built on top of an RTOS, which launches at boot, and firmware that can host multiple applications. These applications could range from providing user interaction through a shell or launching several instances of the same application by means of invoking multiple threads. Not every RTOS toolchain came prepackaged with enough example applications that fit all categories, but to establish a ground truth, we estimated that this was unnecessary. As such, some firmware images built on top of different RTOSs will host similar applications. A full description of each image can be seen in table 5.1. In this table, we have grouped firmware images by their corresponding RTOS. This is followed by their file names and properties, such as optimization flag, MCU manufacturer, Cortex-M revision, and shipped application(s). All images that were used to establish a ground truth can be found in the `fw-dataset` directory on the GitHub page[8].

The custom dataset allows us to establish a ground truth, which will be used to estimate the correctness of the approach described in chapter 4. This dataset will also be utilized to test whether or not our approach is effective and establish the accuracy of PinDown. We can determine what the addresses of application code are as we are in possession of non-stripped binaries as well as the source code of the compiled applications. This makes it easier to find the application code in the compiled binaries using manual analysis and reverse engineering. Using this information, we can correctly verify the results of the analysis performed by PinDown.

Table 5.1: General information for each binary firmware image in the ground truth dataset. The images in question are grouped per RTOS.

Firmware Information \ Properties		Optimization	MCU Manufacturer	Cortex-M Version	Application
RIOT	nordic-riot-nrf52dk.bin	O0	Nordic	M4	Blinky
	nordic-riot-nrf5340dk-app.bin	O5	Nordic	M33	Blinky
	ti-riot-cc1352-launchpad.bin	O0	TI	M4	HelloWorld
	ti-riot-cc2538dk.bin	O5	TI	M3	Micropython
Zephyr	nordic-zephyr-nrf51dk.bin	O5	Nordic	M0	PWM LED Controller
	nordic-zephyr-nrf52840dongle.bin	O5	Nordic	M4	Bluetooth LED Controller
	nordic-zephyr-thingy52.bin	O5	Nordic	M4	HelloWorld
	ti-zephyr-cc1352r-sensortag.bin	O5	TI	M4	Multithreaded blinky
mBed OS	nordic-mbedos-epatlas.bin	O5	Nordic	M4	DeviceKey
	nordic-mbedos-NRF52DK.bin	O5	Nordic	M4	Blinky
	nordic-mbedos-SDT52832B.bin	O5	Nordic	M4	KeyValueStorage
Contiki-NG	nordic-contiki-nrf5340dk.bin	O0	Nordic	M33	Listener
	ti-contiki-cc26x0-http.bin	O0	TI	M4	Client
	ti-contiki-cc26x0-ws.bin	O0	TI	M4	Server
	ti-contiki-cc2538dk.bin	O5	TI	M3	Client
LittleKernel	nordic-lk-nrf51-pca10028-O5.bin	O5	Nordic	M0	Command shell
	nordic-lk-nrf52-pca10040.bin	O0	Nordic	M4	Command shell
	nordic-lk-nrf52-pca10040-O5.bin	O5	Nordic	M4	Command shell
	ti-lk-lm3s6965evb-test.bin	O0	TI	M3	Command shell
Nuttx	nordic-nuttx-nrf52832-dk.bin	O5	Nordic	M4	Button controls, Command shell
	nordic-nuttx-nrf52840-dk.bin	O5	Nordic	M33	PWM controls, Command shell
	ti-nuttx-eagle100.bin	O5	TI	M4	Network Client
	ti-nuttx-lm3s6965-ek.bin	O5	TI	M3	Echo server
	ti-nuttx-tm4c129e-launchpad.bin	O5	TI	M3	Command shell, IPv6 pinger

[8]PinDown Github: `https://github.com/utwente-scs/firmware-decomposition`

5.2.2. Ground truth validation: Stage 1

Ground truth validation was performed on the stripped `.bin` files, which were generated using installed toolchains. First, we need to determine if FirmXRay and Heapster have identified the correct properties of the binary in question. This includes the base address, binary entry point, and function addresses of basic heap modifying functions. The results of this analysis show that, indeed, FirmXRay was able to identify the correct base address for each firmware sample in the ground truth dataset. We verified the correctness of FirmXRay by cross-referencing the identified base address with the base address in the corresponding `.elf` file for each sample.

When FirmXRay is finished, we execute Heapster to identify the addresses of heap modifying functions. When this process is finished, we verify the correctness of the identified functions by checking if the targets in the `.bin` sample match the function models provided by Heapster. Using these models, we concluded that Heapster correctly identified each function for which a model was provided unless that function was not present in the binary.
Following the execution of FirmXRay and Heapster, we need to establish if a function that creates threads has been found. If such an address has been identified, it can be found within the list of caller candidates as a result of the first stage of analysis. We will verify the results of this analysis phase by manually identifying the function address of interest within the corresponding sample. Next, we will cross reference the presence of this address in the acquired list of caller candidates resulting from this analysis phase. As shown in table 5.2, we find that, generally, the first stage of analysis successfully identifies a candidate function that creates threads. We conclude that in these successful cases, reaching definitions analysis can identify a function that creates threads as a caller candidate. We also find that this phase does not identify a correct candidate for a function that creates threads in several other samples. The results show that these samples belong to the groups of mBed OS, Contiki-NG, and RIOT-based samples.

We want to know why these functions do not get identified. When inspecting the samples of mBed OS, we can quickly deduce that applying reaching definitions analysis to the basic functions identified with Heapster does not allow the intended targets to be reached. This can be attributed to the incorrect identification of functions. While this does not appear to be the case in Ghidra, inspection of the stripped binaries in ANGR shows that the addresses of functions that need to be reached are not present in the knowledge base as they have not been recognized as proper functions. As such, the potential callers have not been reached, and the analysis has failed. All samples compiled with Contiki-NG fail to pass stage 1 analysis as well. For both Contiki-NG and mBed OS, this is mainly attributed to the incorrect identification of functions that need to be reached before we arrive at a candidate function that creates a thread. The functions at which reaching definitions analysis terminates prematurely varies between binaries. We have deduced that this can be attributed to the toolchain applying optimizations based on the target hardware. The actual function that creates threads is, however, disassembled properly, so if we were able to successfully complete reaching definitions analysis, the candidate would be present in the resulting list. Lastly, our analysis also seems to fail in successfully identifying caller candidates in RIOT-based samples. As with mBed OS and Contiki-NG, we find that intermediate functions in the reaching definitions analysis are not correctly identified or that the caller candidate we aim to identify is improperly disassembled, causing the analysis process to terminate prematurely and not reach the intended target. The issue of improper disassembly and misidentification of functions is well known within the field of reverse engineering, and as such, this seems to be a limitation of the technology we utilize and not necessarily our approach or implementation. This is supported by the fact that our approach to identifying these caller candidates is effective in the samples that host the other operating systems. We have accounted for this problem by integrating an alternative path of application code discovery, as we are aware of the possibility that functions may not be correctly identified.

5.2.3. Ground truth validation: Stage 2

Next, we will investigate the results acquired from the second analysis stage. Our approach successfully identified the functional component and corresponding target function in 21 samples, failing in only four cases, as shown in table 5.2. A closer inspection of the four cases that fail stage 2 analysis shows that the samples that fail analysis are all built on top of the Contiki-NG RTOS. When inspecting the logs and debugging information we generated during analysis, we found that our implementation analyzed a function that creates threads in every sample that failed during stage 2 analysis. This means that our approach was able to identify the correct function based on our alternative path of caller candidate

Table 5.2: An overview of whether or not stage 1 and stage 2 analysis were completed successfully, whether the firmware was disassembled correctly as well as how many candidates were found at the end of stage 2 with the corresponding false positive rate.

Firmware Name	*Stage 1 Success*	*Correct Disassembly*	*Stage 2 Successful*	*Number Of Identified Functions*	*Stage 2 False Positive Rate*
nordic-riot-nrf52dk.bin	-	-	+	1	0%
nordic-riot-nrf5340dk-app.bin	-	-	+	2	50%
ti-riot-cc1352-launchpad.bin	-	+	+	1	0%
ti-riot-cc2538dk.bin	-	-	+	2	50%
nordic-zephyr-nrf51dk.bin	+	+	+	2	50%
nordic-zephyr-nrf52840dongle.bin	+	+	+	1	0%
nordic-zephyr-thingy52.bin	+	+	+	3	66.7%
ti-zephyr-cc1352r-sensortag.bin	+	+	+	3	66.7%
nordic-mbedos-epatlas.bin	-	-	+	1	0%
nordic-mbedos-NRF52DK.bin	-	-	+	1	0%
nordic-mbedos-SDT52832B.bin	-	-	+	1	0%
nordic-contiki-nrf5340dk.bin	-	+	-	2	50%
ti-contiki-cc26x0-http.bin	-	-	-	2	50%
ti-contiki-cc26x0-ws.bin	-	-	-	2	50%
ti-contiki-cc2538dk.bin	-	+	-	2	50%
nordic-lk-nrf51-pca10028-O5.bin	+	+	+	1	0%
nordic-lk-nrf52-pca10040-bin	+	+	+	2	0%
nordic-lk-nrf52-pca10040-O5.bin	+	+	+	1	50%
ti-lk-lm3s6965evb-test.bin	+	+	+	2	50%
nordic-nuttx-nrf52832-dk.bin	+	+	+	1	0%
nordic-nuttx-nrf52840-dk.bin	+	+	+	1	0%
ti-nuttx-eagle100.bin	+	+	+	1	0%
ti-nuttx-lm3s6965-ek.bin	+	+	+	1	0%
ti-nuttx-tm4c129e-launchpad.bin	+	+	+	2	50%

discovery despite the failure of the first stage of analysis. During the retrieval of register values and loaded values, we observed that no valid function addresses were found despite manual analysis of instructions indicating that a proper function address was loaded from memory. It is unexpected that this analysis stage fails as we take into account the edge case possibility that function addresses can be part of a list data structure or that such addresses are contained in pointers. However, closer inspection reveals that these function addresses are loaded into virtual memory tables, and the addresses are inaccessible during static analysis. This virtual address table is never constructed, as we are not executing or attempting to parse the generation of such virtual address tables. As such, our analysis could not discover valid functions within these binary samples, so the analysis was terminated unsuccessfully. Our implementation and approach would have been able to successfully identify the function arguments if these were not stored in a virtual address table, as our approach was able to reach the function responsible for creating threads despite stage 1 analysis failing. Furthermore, despite several cases failing to identify the intended caller candidate, the alternative discovery implementation shows that functional components can still be identified even if we were not able to find proper candidates using reaching definitions analysis. This is done by leveraging knowledge about the call graphs of sample

binaries and how functions are passed as arguments to construct a generalized approach. While this alternative discovery has the potential to introduce more false positives, results show that this is not a guaranteed outcome as PinDown was able to identify the functional component of importance in samples on which stage 1 analysis was unsuccessful, realizing a false positive rate on par with samples in which stage 1 was successful. This indicates that within the group of selected RTOSs, stage 1 analysis may be redundant and that the heuristics-based method of identifying functions that create threads is sufficient for this selection.

5.2.4. Ground truth validation: Stage 3

For the third stage of analysis, we find that the results for all samples match our expectations while being generally promising, as seen in table 5.3. The fact that stage 3 analysis fails for all samples with Contiki-NG is unsurprising, as the second analysis stage was unsuccessful for these samples. This results in there not being any candidates that could be analyzed during the third analysis stage. PinDown analysis was, however, successful in identifying the relevant function(s) that could host application code in every other sample from the ground truth dataset. While it is a rare occurrence that exactly and only the function addresses are identified which host application code, this is expected based on the approach we have implemented and given the fact that static analysis generally has a higher false positive rate compared to other analysis techniques. Based on the results of the third analysis stage, we can conclude that although multiple candidates were identified, there is still a significant reduction in candidate functions. The amount of candidate application code functions is reduced from several hundred to an average of only five candidate functions. Even without the context of the device's intended application, it is relatively easy to identify which addresses contain application code versus which addresses perform RTOS-related functions. This is because many of the incorrectly identified functions seem to perform memory-related operations and are quite short in terms of function size. Moreover, the implemented approach was always able to identify the addresses of user applications and the associated data structure in which these addresses may be stored. When the applications are stored in a data structure such as a list, a quick analysis of the memory regions where the references are stored reveals whether or not other applications might be hosted on a system. However, as such lists may be constructed in a different manner (such as different separators for each list entry), we have not implemented an approach to automatically identify all these function addresses and properties as this would require knowledge of the data structure at hand. As mentioned previously, we assume no knowledge of a present RTOS and cannot make any assumptions about the implemented data structures. This means that PinDown was successful in identifying application code in 84% of the samples in the ground truth dataset, failing only when application code was stored at an unreachable address. This is a limitation of the implementation as we want to keep the approach static.

The results of our ground truth validation clearly show that overall analysis can successfully terminate and correctly identify the application code and functional components within a firmware image. This is achieved despite some failures of the intermediary analysis phases. PinDown reduced the number of candidate functions that could host application code in samples of our ground truth dataset by 99.18% on average and shows a false positive rate of 50.26% on average when inspecting the selected candidate functions. While this rate is rather high in practice, there are less than ~ 5 candidate functions identified that could host application code. This is an acceptable number of functions to manually investigate when one needs to determine the application code of a device.

5.3. Experimentation: Wild firmware

The experiments on the wild firmware dataset were performed using the same experimental setup as with the ground truth dataset. The results were validated by manual verification of the identified addresses, using the same approach as in the validation of our ground truth dataset.

5.3.1. Wild firmware dataset description

The results from the ground truth validation show that the implemented approach is effective in identifying application code. The next goal is to assess its effectiveness when applied to wild firmware. In this instance, we collected firmware samples from various collections. A selection of samples was curated from several collections of monolithic firmware that matched our scope. The dataset includes

Table 5.3: Here is highlighted in which cases stage 3 was successful followed by the rate by which we were able to reduce the candidate functions in a binary firmware image.

Firmware Name	*Stage 3 Successful*	*Number of Identified Functions*	*Stage 3 False Positive Rate*	*Candidate Functions*	*Candidate Reduction Rate (%)*
nordic-riot-nrf52dk.bin	+	2	50%	2/197	98.99%
nordic-riot-nrf5340dk-app.bin	+	6	83.3%	6/189	96.83%
ti-riot-cc1352-launchpad.bin	+	3	66.7%	4/175	97.71%
ti-riot-cc2538dk.bin	+	3	66.7%	3/2046	99.85%
nordic-zephyr-nrf52840dongle.bin	+	6	83.3%	6/583	98.97%
nordic-zephyr-nrf51dk.bin	+	7	85.75%	7/478	99.68%
nordic-zephyr-thingy52.bin	+	8	87.5%	8/394	99.48%
ti-zephyr-cc1352r-sensortag.bin	+	2	50%	2/266	99.25%
nordic-mbedos-epatlas.bin	+	3	66.7%	3/1043	99.71%
nordic-mbedos-NRF52DK.bin	+	3	66.7%	3/549	99.45%
nordic-mbedos-SDT52832B.bin	+	3	66.7%	3/543	99.45%
nordic-contiki-nrf5340dk.bin	-	0	100%	736/736	0%
ti-contiki-cc26x0-http.bin	-	0	100%	823/823	0%
ti-contiki-cc26x0-ws.bin	-	0	100%	835/835	0%
ti-contiki-cc2538dk.bin	-	0	100%	681/681	0%
nordic-lk-nrf51-pca10028-O5.bin	+	2	50%	2/486	99.59%
nordic-lk-nrf52-pca10040-bin	+	3	66.7%	3/488	99.39%
nordic-lk-nrf52-pca10040-O5.bin	+	1	0%	1/529	99.81%
ti-lk-lm3s6965evb-test.bin	+	4	75%	4/605	99.34%
nordic-nuttx-nrf52832-dk.bin	+	2	50%	2/979	99.79%
nordic-nuttx-nrf52840-dk.bin	+	2	50%	2/972	99.78%
ti-nuttx-eagle100.bin	+	3	66.7%	3/716	99.57%
ti-nuttx-lm3s6965-ek.bin	+	1	0%	1/932	99.89%
ti-nuttx-tm4c129e-launchpad.bin	+	4	75%	4/1354	99.70%

a firmware samples from the collection ucsb-seclab[9].

Samples were selected based on whether or not they fit the previously established scope. Other characteristics, such as image size or intended application, were not considered during the selection, and the samples were not manually inspected or reverse-engineered before performing the experiment. As with the ground truth dataset, a similar description of each sample in the wild firmware dataset has been provided in table 5.4. There is little known about each sample regarding how it was compiled or what the target hardware was. While we could derive some characteristics, it is unrealistic to derive the missing information accurately. In the cases where we could not derive certain properties, **U** was used to convey an ”unknown” property. We also performed an experiment on an image containing an operating system that had not been inspected or analyzed before. This information was used to verify if our approach is also effective in identifying application code within firmware hosting an operating system that had not been used in establishing our ground truth. Given the scarce availability of wild firmware that can be analyzed, these results indicate the effectiveness of the approach. In total, we inspect images based on RIOT OS, Zephyr OS, mBed OS, Contiki-NG, and FreeRTOS.

5.3.2. Wild firmware validation: Stage 1

We applied FirmXRay and Heapster to each image in the dataset, as we did with samples from the ground-truth dataset. The analysis of each image successfully terminated. FirmXRay correctly iden-

[9]monolithic firmware collection:`https://github.com/ucsb-seclab/monolithic-firmware-collection`

Table 5.4: General information for each binary firmware image in the wild firmware dataset. The images in question are grouped per operating system and a **U** is used to describe an unknown property.

Firmware Information \ Properties		Optimization	MCU Manufacturer	Cortex-M Version	Application
RIOT	ti-p2im_console.bin	O0	TI	**U**	Command shell
Zephyr	zephyr-CVE-2020-10064.bin	**U**	**U**	**U**	Echo server
	zephyr-CVE-2020-10065.bin	**U**	**U**	**U**	Bluetooth device
	zephyr-CVE-2020-10066.bin	**U**	**U**	**U**	Bluetooth device
	zephyr-CVE-2021-3320.bin	**U**	**U**	**U**	Echo server
	zephyr-CVE-2021-3321.bin	**U**	**U**	**U**	Echo server
	zephyr-CVE-2021-3322.bin	**U**	**U**	**U**	Echo server
	zephyr-CVE-2021-3323.bin	**U**	**U**	**U**	Echo server
	zephyr-CVE-2021-3329.bin	**U**	**U**	**U**	Bluetooth device
	zephyr-CVE-2021-3330.bin	**U**	**U**	**U**	Echo server
	CVE-no-CVE-false-positive-watchdog-callback.bin	**U**	**U**	Cortex-M	Echo server
	zephyr-CVE-no-CVE-false-positive-rf-size-check.bin	**U**	**U**	Cortex-M	Echo server
mBed OS	mBed-arch-pro.bin	**U**	NXP	M3	Secret code reader
	mBed-efm32gg-stk3700.bin	**U**	Silicon labs	M3	Secret code reader
Contiki-NG	contiki-atmel_6lowpan_udp_rx.bin	**U**	**U**	**U**	UDP receiver
	contiki-atmel_6lowpan_udp_tx.bin	**U**	**U**	**U**	UDP transmitter
	contiki-cve-2020-12140.bin	**U**	**U**	**U**	UDP receiver
FreeRTOS	freertos-p2im_soldering_iron.bin	**U**	**U**	**U**	Soldering iron controls

tified all base addresses, which were then used to find the entry points within each binary. Heapster was then able to find the addresses of all modelled heap-modifying functions present in the binary.

Given the results from the execution of FirmXRay and Heapster, we now initiate the caller candidate identification. Unfortunately, we quickly find that reaching definitions analysis does not seem effective at identifying functions that create threads, as can be seen in table 5.5. We see that such a function is not identified in the firmware based on RIOT OS, mBed, Contiki-NG, and FreeRTOS. This means that this approach was only effective at identifying such a function within Zephyr-based firmware. Closer inspection indicates that this can only be partially blamed on the incorrect identification of functions. Furthermore, the actual functions that implement thread creation seem to be disassembled properly in all samples except those based on mBed OS. Inspecting the Contiki-NG and FreeRTOS-based firmware indicates that these implementations do not depend on HML components to accommodate functions for which threads are created. Instead, they use different means of achieving this. We also find that the functions that implement thread creation are important to the initialization of user applications, so this implies that our method of identifying such functions was made with an incorrect assumption. The alternate path of function discovery was implemented to remedy this potential flaw, which was shown to be effective in our ground truth dataset. Regarding Zephyr OS, we can correctly identify candidate functions that create threads.

The acquired results from stage 1 analysis show that for our approach to work, as expected, we require a set of conditions to be true. These conditions indicate the extent to which our approach is applicable to different RTOSs. These conditions are:

- The function tasked with creating and executing threads should be correctly identified as a function.
- The function tasked with creating and executing threads must leverage HML functions to be identified.
- Each function that is analyzed during reaching definitions analysis is required to be correctly identified as a function.

5.3.3. Wild firmware validation: Stage 2

After the first stage of analysis is finished, we initiate stage 2 analysis. Our approach was able to correctly identify a candidate function that implements thread creation in 12 firmware samples. As we can see in table 5.5, we find that our implementation was able to identify the functional components

Table 5.5: An overview of whether or not analysis phases were completed successfully, whether the firmware was disassembled correctly as well as how many candidates were found at the end of stage 2 analysis with the corresponding false positive rate.

Firmware Name	*Stage 1 Success*	*Correct Disassembly*	*Stage 2 Successful*	*Number Of Identified Functions*	*Stage 2 False Positive Rate*
ti-p2im_console.bin	-	+	+	1	0%
zephyr-CVE-2020-10064.bin	+	+	+	1	0%
zephyr-CVE-2020-10065.bin	+	+	+	2	50%
zephyr-CVE-2020-10066.bin	+	+	+	2	50%
zephyr-CVE-2021-3320.bin	+	+	+	1	0%
zephyr-CVE-2021-3321.bin	+	+	+	1	0%
zephyr-CVE-2021-3322.bin	+	+	+	1	0%
zephyr-CVE-2021-3323.bin	+	+	+	1	0%
zephyr-CVE-2021-3329.bin	+	+	+	2	50%
zephyr-CVE-2021-3330.bin	+	+	+	1	0%
false-positive-watchdog-callback.bin	+	+	+	1	0%
false-positive-rf-size-check.bin	+	+	+	1	0%
mBed-arch-pro.bin	-	-	+	4	25%
mBed-efm32gg-stk3700.bin	-	-	+	4	25%
contiki-atmel_6lowpan_udp_rx.bin	-	+	-	0	100%
contiki-atmel_6lowpan_udp_tx.bin	-	+	-	0	100%
contiki-cve-2020-12140.bin	-	+	+	10	90%
freertos-p2im_soldering_iron.bin	-	+	-	2	100%

within the Zephyr, RIOT, and mBed firmware samples despite failing stage 1 analysis. Furthermore, we find that our implementation identified candidate functions in one Contiki-NG and one FreeRTOS-based binary firmware image, but the identified candidates in the FreeRTOS sample are all incorrect. This means that our alternate path of discovery was able to identify functions that matched the requirement that it should have a valid function address as an argument in the higher levels of the call graph.

A closer look at the structure of the binary image based on FreeRTOS reveals that thread creation is not important to the initialization of user application code. Instead, the main user application is called directly. This results in our approach being unable to identify relevant functions as they are simply absent. Due to the difficulty of discerning between user applications and functions introduced by the framework, our implementation could not leverage previously discussed concepts to identify application code within this image. When manually analyzing functions that create threads within this image, we find that the arguments passed to a function that creates threads are again pointers to the heads of lists stored within memory. Unfortunately, the structure of these lists is different from the ones we accounted for in our implementation. As such, these functions were not able to be identified through the path of alternate discovery. However, this would not matter as these functions were called within the user application code, and as such, the arguments would have been incorrectly identified as functional components. This indicates the extent to which our approach is applicable to different RTOSs.

The candidates from the Contiki-NG-based image are more interesting as we could not identify any valid candidates in the images that were used to establish a ground truth. Manual analysis of this image shows that PinDown could identify a function that starts user application processes, but not in the way we expected. Further inspection shows that the arguments passed to this component are the head elements of lists stored in memory. In our ground-truth dataset, we dealt with addresses that could not

be reached as they were stored in virtual address tables. While the addresses in this current Contiki-NG sample can be reached, the list structure shows index separators that we had not accounted for in the implementation. This means that the functional component will not be identified and classified as such based on this analysis. Instead, we find that the functional component has been identified based on another function that calls a function that creates threads at a different point in the call graph. This means that while the function tasked with executing user applications or a functional component has been identified, the point at which it was identified plays no role in the initialization of user application code. This means that the next analysis phase will not be able to identify any relevant application code for this sample. If we had been able to account for different data structures that store user application addresses, we would have identified the correct functional component or user application.

5.3.4. Wild firmware validation: Stage 3

The third stage of analysis will take candidate functions identified in the second analysis stage. These candidate functions will be used to find user application code, the results of which can be viewed in table 5.6. We find that our approach correctly identified user application code in the samples based on Zephyr and RIOT OS. We also found that our approach was unsuccessful in identifying user application code in the samples based on mBed OS, Contiki-NG, and FreeRTOS. As discussed in the previous section, the failure to identify application code in Contiki-NG and FreeRTOS-based binary firmware images was expected due to the lack of accounting for the data structure that required parsing. The fact that our approach failed to identify the application code within the mBed OS-based firmware is more surprising and requires a more thorough analysis.

The functions identified in mBed OS-based samples at the end of stage 2 analysis show that the components are the same as the components found within the mBed OS-based samples within the ground truth dataset. This means that we could identify a function that is important to the process of initializing application code. We will manually reverse the binary image to identify what went wrong. After the manual analysis is complete, it becomes clear that the functional component that is leveraged in order to identify application code is structurally different from the one we identified within the ground truth dataset for reasons unknown. A disassembly of the critical function that was identified in these images can be seen in figure 5.1a. After manual analysis, we determined that the same function was identified in both wild mBed OS-based samples. However, the call targets of this function are no longer present, as can be seen in figure 5.1b. This means that the critical component was correctly identified in a generalized manner, but the analysis could not find any call targets as these were absent. This means that analysis had to halt and that no user application code could be identified. Unfortunately, we do not know why this critical function differs from the one found in images from our ground truth dataset. Deriving the cause of this difference is also difficult as we do not have any knowledge about how it was generated, the version of the toolchain, or the version of the compiler that was used. We are also unaware of whether these factors are of any importance to begin with. It could be that the function was removed or altered by developers or that the toolchain optimization stripped it from the final image. Another possibility is that the function has been altered because the image was disassembled incorrectly. Developers may be inclined to apply changes as they see fit, but such changes are challenging to differentiate from issues arising from firmware reversing. The result is that PinDown will not be able to analyze images as analysis is based on the identification of mechanisms within unmodified operating systems. This is a limitation of the approach, which is difficult to make up for, given how we cannot make any assumptions about how developers decide to alter operating system functionalities.

When analyzing wild firmware, we found that PinDown was able to reduce the number of candidate functions by 99.47%. Furthermore, we have found that among the identified candidates, there exists an average false positive rate of 75.08%. However, we find that the result of the final analysis stage generally contains the function addresses that host user applications. Unfortunately, given how wild binary firmware images are hard to come by, we find ourselves somewhat constrained given the scope of the project. Furthermore, we were able to identify features that made it impossible for our approach to correctly identify user application code.

5.4. Performance

In order to determine the applicability of the PinDown framework in real-life analysis and understand the limitations of the research, we have logged the execution times of each analysis stage. Inspect-

```
void software_init_hook(void)

{
  mbed_stack_isr_start = &__StackLimit;
  mbed_stack_isr_size = 0x400;
  mbed_heap_start = &end;
  mbed_heap_size = 0x3a2f0;
  mbed_init();
  mbed_rtos_start();
                    /* WARNING: Bad instruction - Truncating control flow here */
  halt_baddata();
}
```

(a) The annotated disassembly of the critical component identified in the `nordic-mbedos-epatlas.bin` image.

```
void software_init_hook_rtos(void)

{
  return;
}
```

(b) The annotated disassembly of the critical component identified in the `mbed-arch-pro.bin` image.

Figure 5.1: Annotated disassemblies of the same function within two different binary firmware images based on mBed OS.

ing the performance of the analysis framework indicates that the speed of the framework is bound by the analysis performed by Heapster as well as the *caller candidate discovery* step in the first analysis stage. We also find that overall performance is impacted by the present user application(s), RTOS, and changes that developers have applied to the said RTOS framework. Furthermore, the performance of the second and third analysis stages are bound by the implemented workarounds that aim to address the challenges mentioned in chapter 4. This can be mainly attributed to the fact that when a potential infinite loop is encountered, we wait a constant amount of iterations before we decide to halt the symbolic execution and continue analysis. We also find that the amount of time required by the third analysis stage is nearly negligible in all firmware images within both the wild firmware and ground truth datasets. The execution times also show that when earlier analysis stages are unsuccessful, future stages are impacted in their relative performance. This is expected as when analysis fails to identify specific targets, more possible targets are taken into account, which leads to an increasing number of functions being analyzed. Finally, we can observe that the total time necessary for PinDown to analyze a binary firmware image is about 23.5 minutes, which is an acceptable amount of time, especially given how this process can be easily parallelized, which will greatly reduce the downtime between analyzing several firmware images at once.

5.4.1. Ground truth dataset

We start with investigating the performance results of the entire PinDown framework, of which the results are shown in figure 5.2. It becomes evident that the prerequisite analysis phase generally causes the majority of the performance overhead. We also find that, on average, PinDown is responsible for about 7.7% of the performance overhead. This is in line with our expectations, as PinDown relies mainly on static analysis techniques and the symbolic execution of only a few specific functions. The number of functions that PinDown will analyze will be decided by the prerequisite analysis phase, which will need to create a selection of all candidates for which static analysis and function simulation will be utilized. We can also see that there is one statistical outlier in `ti-riot-cc2538dk.bin`, which we will inspect in greater detail.

We highlight the performance results of the individual processes within the prerequisite analysis phase, which can be viewed in figure 5.3. It becomes very clear that the performance of the first stage of analysis is bound by the execution time of Heapster and the caller candidate identification. This was expected as Heapster simulates functions it aims to identify, meaning it will symbolically execute each function in a binary firmware image if the function was not filtered out based on model expectations. As the caller candidate discovery iterates through the call graph, starting at each basic HML function, there will be multiple paths to traverse. These paths will all need to be analyzed, and given the complexity of RTOS, many paths will greatly impact performance. The possibility of functions being incorrectly identified may also lead to the analysis framework exhaustively trying to identify a potential generator. Furthermore, the processes of caller candidate identification were combined with the pointer source generator identification step from Heapster in order to reduce potential overhead, all of which explain the long execution times of stage 1 analysis as well as why stage 1 analysis represents 64.1% of the average performance overhead.

This is further supported by derived metrics as shown in table 5.8 in which we find an average execution time of 1300.09 seconds, of which 833.9 seconds on average are spent performing caller

Table 5.6: Additional results showing the success of stage 3 analysis and the reduction rate of candidate functions in various firmware images.

Firmware Name	*Stage 3 Successful*	*Number of Identified Functions*	*Stage 3 False Positive Rate*	*Candidate Functions*	*Candidate Reduction Rate (%)*
ti-p2im_console.bin	+	3	66.7%	3/336	99.11%
zephyr-CVE-2020-10064.bin	+	3	66.7%	3/910	99.67%
zephyr-CVE-2020-10065.bin	+	6	83.3%	6/816	99.26%
zephyr-CVE-2020-10066.bin	+	5	80%	5/832	99.40%
zephyr-CVE-2021-3320.bin	+	4	75%	4/889	99.55%
zephyr-CVE-2021-3321.bin	+	4	75%	4/885	99.55%
zephyr-CVE-2021-3322.bin	+	4	75%	4/886	99.55%
zephyr-CVE-2021-3323.bin	+	4	75%	4/882	99.55%
zephyr-CVE-2021-3329.bin	+	5	80%	5/843	99.41%
zephyr-CVE-2021-3330.bin	+	4	75%	4/876	99.55%
zephyr-CVE-no-CVE-false-positive-watchdog-callback.bin	+	4	75%	4/890	99.55%
zephyr-CVE-no-CVE-false-positive-rf-size-check.bin	+	4	75%	4/896	99.55%
mBed-arch-pro.bin	-	0	100%	391/391	0%
mBed-efm32gg-stk3700.bin	-	0	100%	484/484	0%
contiki-atmel_6lowpan_udp_rx.bin	-	0	100%	804/804	0%
contiki-atmel_6lowpan_udp_tx.bin	-	0	100%	804/804	0%
contiki-cve-2020-12140.bin	-	0	100%	671/671	0%
freertos-p2im_soldering_iron.bin	-	0	100%	689/689	0%

Table 5.7: Statistics for the prerequisite analysis, PinDown analysis and combined total performances where the combined performance is based on the sum of respective execution times.

	Combined execution time	**Prerequisite analysis**	**PinDown analysis**
Statistic	Statistcal values		
Average	1408.20	1300.09	108.11
Minimum	162.46	104.92	5.54
Maximum	4984.80	4853.81	1026.80
Standard Deviation	1003.03	978.32	217.81
Median	1507.31	1437.85	28.57
Variance	1006065.54	957112.43	47440.16
Number of Outliers	1	1	1

candidate discovery. This is in line with our expectations as tasks relayed to Heapster and caller candidate discovery are more computationally intensive to complete compared to the task of FirmXRay. We also find that `ti-riot-cc2538dk.bin` is the only statistical outlier that varies more than twice the standard deviation from the average total execution time. Closer inspection reveals that this can be attributed to the fact that several functions are not correctly identified by `angr`, which leads to several instances within Heapster and caller candidate identification trying to identify functions that it cannot reach despite these being present in the knowledge base. As such, an exhaustive discovery is performed to identify potential generators and candidates. This can be further supported by the results from the experimental validation, which clearly show that stage 1 analysis was unsuccessful due to improper function identification.

The performance results of stage 2 and 3 analysis indicate that this performance is bound by the analysis performed in stage 2, as shown in figure 5.4. This is also in line with our expectations, as all matching call targets need to be examined for valid function addresses and all the functions present

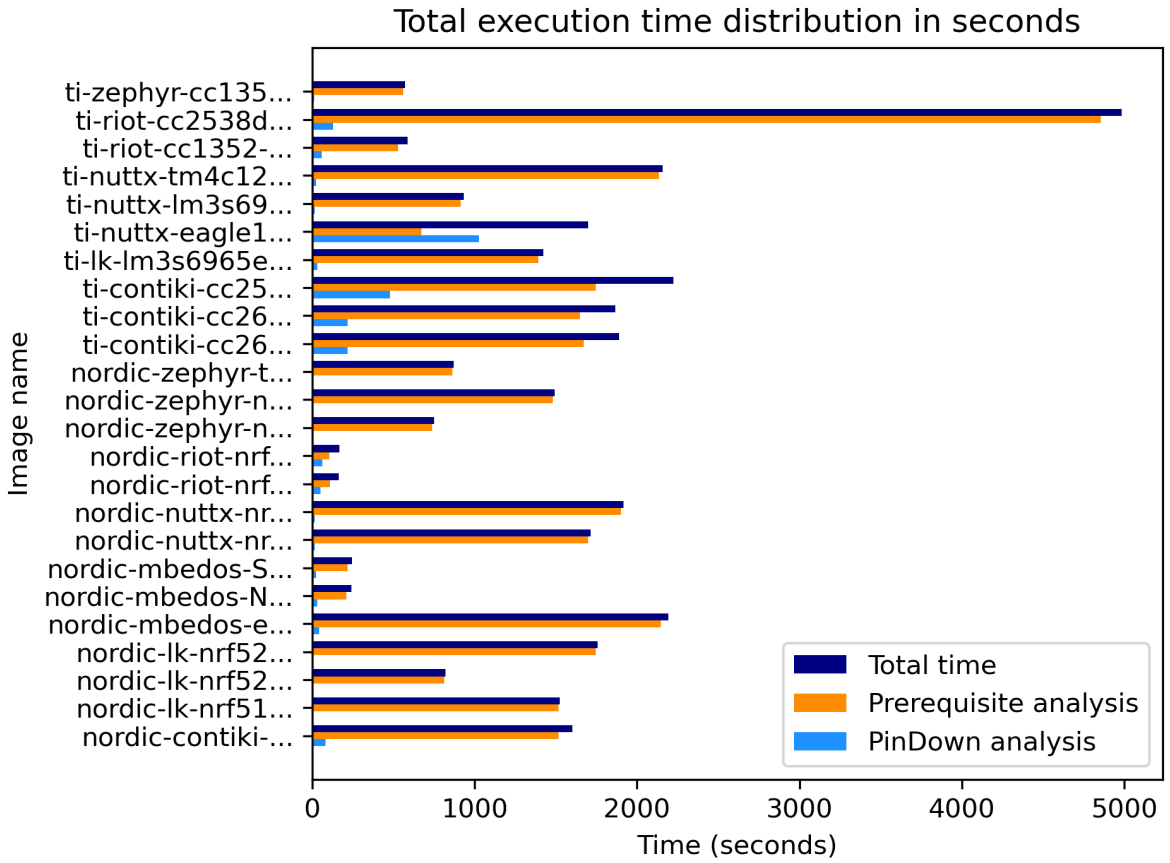

Figure 5.2: The figure shows execution times of each main component in the PinDown analysis framework when applied to every image in the ground truth dataset. It also shows how execution time was distributed between PinDown and the prerequisite analysis phase

in the first three layers of any call graph generated by `angr`. The single statistical outlier of these results is the `ti-nuttx-eagle100.bin` firmware image, which requires a short amount of time to complete the second stage of analysis but then takes comparatively long to complete the third stage of analysis. A closer inspection of this firmware image shows that several infinite loops in the code are analyzed when trying to determine whether the function at hand is application code or not. The image was compiled with a client application that listens continually for possible inputs. It is highly likely that the analysis will take longer than the second stage because of how we deal with potentially infinite loops.

Furthermore, this provides useful insight that the application code present on the image affects the performance of PinDown because of its implementation. We cannot ensure we will never analyze user application code as initialization procedures may call these functions directly. We need to alter our implementation if we want to mitigate the potential performance overhead that the analysis of user applications may introduce.

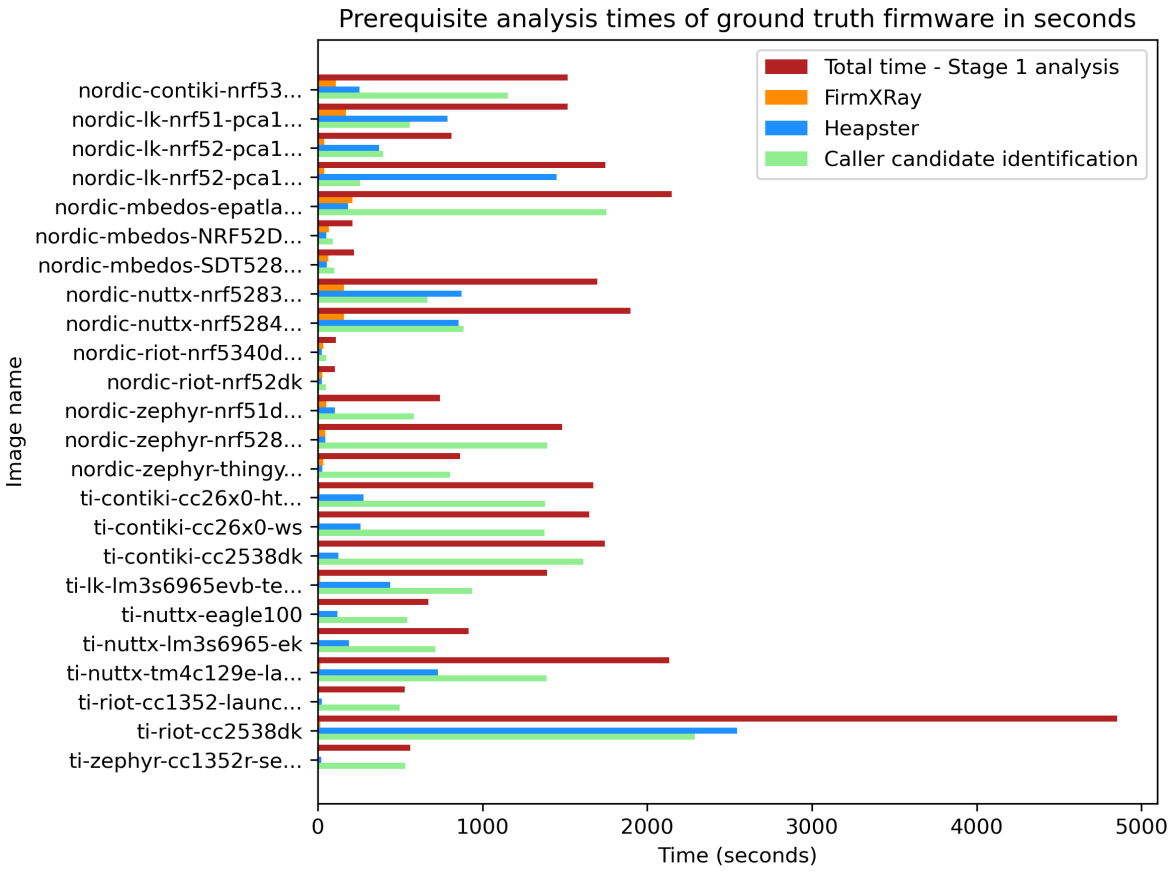

Figure 5.3: Prerequisite performance results of each firmware image in the ground truth dataset in seconds. The total time for each image is equal to the sum of the respective Heapster, FirmXRay and Phase 1 execution times.

Besides the statistical outlier, we also need to consider that the firmware images based on Contiki-NG failed the second stage of analysis. As a result, the analysis process was terminated before the third analysis stage was performed. This explains why stage 2 analysis represents 100% of the total Pin-Down execution time in these cases, which is reflected in the average values shown in table 5.9. When we omit the performance results of `ti-nuttx-eagle100.bin` and `ti-contiki-cc2538dk.bin` (which are the two statistical outliers) to improve the clarity of the graphs as can be seen in figure 5.5, we see that stage 3 has a near negligible share in the performance overhead. This is also apparent based on the generated statistics in table 5.9.

Overall, it becomes clear that the identification of functions that implement thread creation is the bounding factor.

Table 5.8: Combined statistics and metrics on performance results from the ground truth dataset found in figure 5.3. Combined performance corresponds to the sum of the respective FirmXray, Heapster and caller candidate identification execution times.

	Stage 1 performance	FirmXRay performance	Heapster performance	Caller candidate Indentification
Statistic	Statistical values			
Average	1300.09	55.72	410.47	833.90
Minimum	105.00	6.39	21.91	49.84
Maximum	4853.81	211.42	2546.03	2290.53
Standard Deviation	978.32	59.07	572.15	584.81
Median	1437.85	33.91	185.83	691.07
Variance	957112.43	3489.63	327357.60	341998.51
Number of Outliers	1	No Outliers	1	1

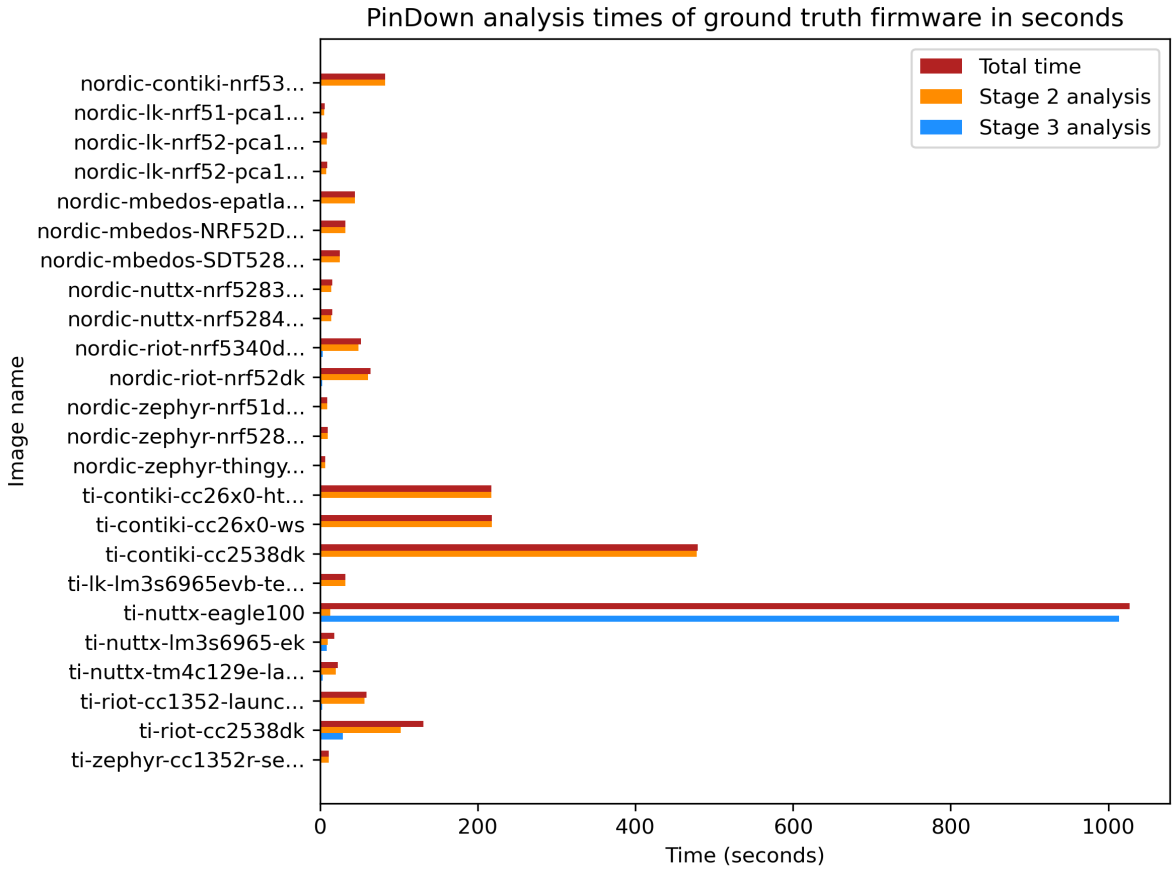

Figure 5.4: Performance results of applying stage 2 and stage 3 analysis to each firmware image in the ground truth dataset. The total time for each image is equal to the sum of the respective stage 2 and stage 3 analysis times.

5.4.2. Wild firmware dataset

We have seen that the first analysis stage has the largest share in performance overhead when analyzing firmware images from the ground truth dataset. We want to know if the same holds for wild firmware or if this was simply a result of how we generated this dataset. Plotting out the execution times, as can be seen in figure 5.6, clearly indicates that the same holds for the analysis of wild firmware. We also find that generally, the average execution times are higher than the ground truth dataset, as seen in table 5.10, showing an increase of nearly 600 seconds on average. Viewing these numbers, we do have to consider that our two datasets are very different regarding the number of images per RTOS, the types of RTOSs present in these images as well as the implemented user applications, all of which impact the performance to some degree. We will now inspect the execution times of all the steps in the first stage of analysis as well as the second and third stages of analysis in order to more accurately determine the main causes of extended execution times. The execution times for the prerequisite analysis phase on wild firmware can be viewed in figure 5.7.

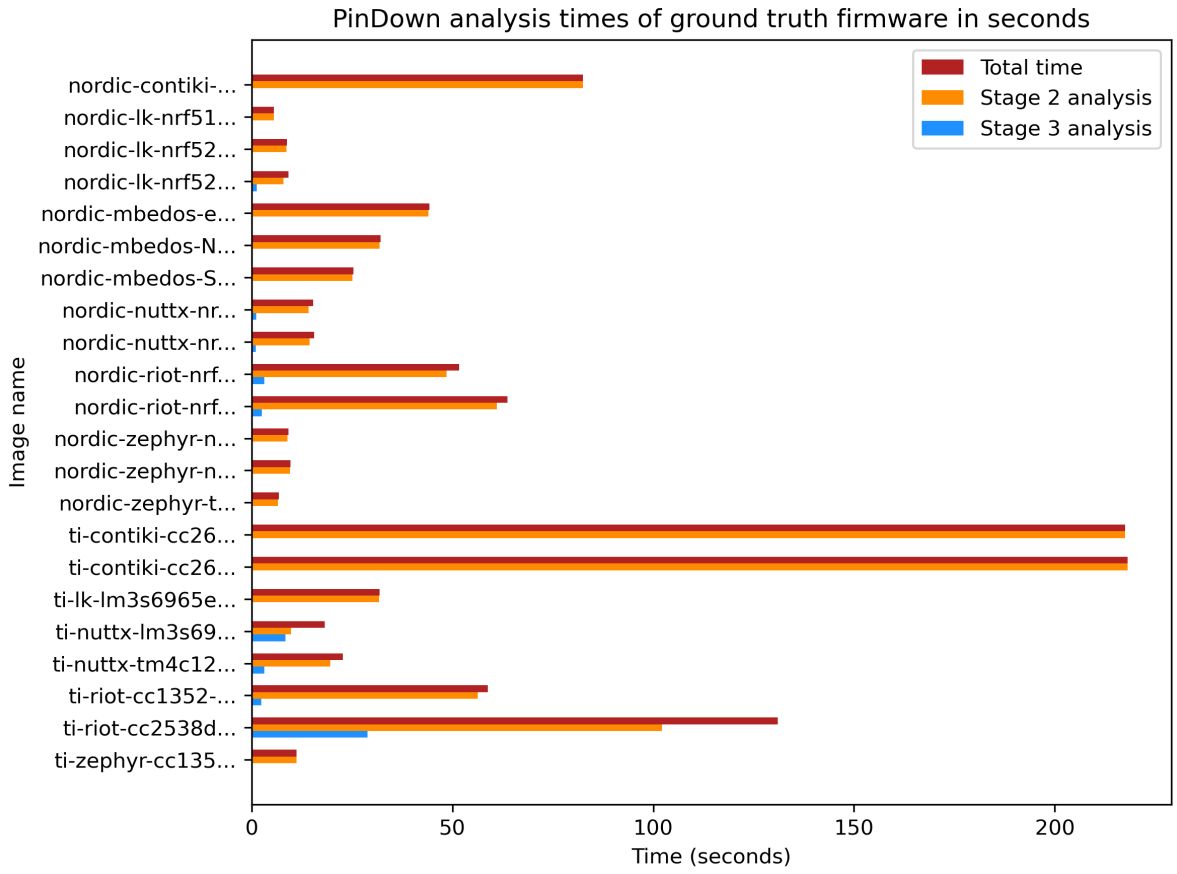

Figure 5.5: Omitting `ti-nuttx-eagle100.bin` and `ti-contiki-cc2538dk.bin` to improve visual clarity.

In our ground truth analysis, we found that the first analysis stage was generally bounded by the identification of the caller candidate. We attribute this to the increasing complexity of reaching definitions analysis and the possibility of incorrectly identified functions. However, when inspecting the performance results concerned with analyzing wild firmware, we find that caller candidate identification only makes up for the majority of the performance overhead in two cases, which are also the statistical maximum values as can be seen in table 5.11. Incidentally, these concern samples whose critical functions do not seem to be properly identified. While this seems to indicate that improper function identification causes extended execution times of caller candidate identification, the results from the ground-truth dataset contradict this notion as several properly disassembled images showed longer execution times of caller candidate identification as well. This means that the operating system impacts the performance of the first stage of analysis, the user application(s) present in the binary firmware image, and how the firmware images were generated. We can derive this based on the fact that images that host a similar operating system have a varying distribution of execution times between Heapster and caller candidate identification. This also implies that it is difficult to give any performance estimations based on the approach and implementation, as performance depends on factors we have no assumed knowledge of.

Table 5.9: Combined statistics for applying PinDown to the ground truth dataset after omitting `ti-nuttx-eagle100.bin` and `ti-contiki-cc2538dk.bin`.

	Total execution time	Stage 2 analysis	Stage 3 analysis
Statistic	Statistical values		
Average	49.50	47.07	2.44
Minimum	5.54	5.49	0.00
Maximum	218.08	218.08	28.86
Standard Deviation	60.91	59.78	6.07

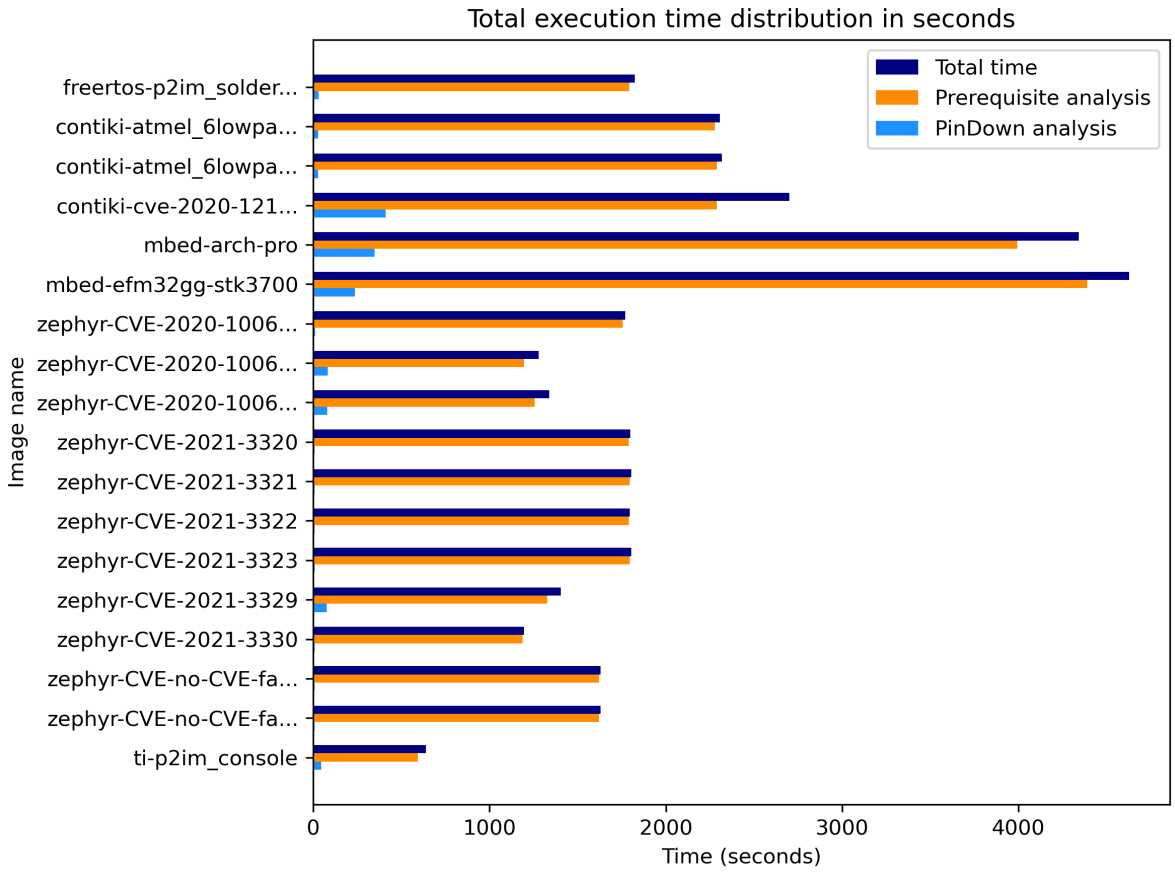

Figure 5.6: The figure shows execution times of each main component in the PinDown analysis framework and how these compare to the total execution time for each image in the wild firmware dataset.

Table 5.10: Statistics for the performance results of applying PinDown to wild firmware.

	Combined Execution Time	Prerequisite Analysis	PinDown Analysis
Statistic	Statistical values		
Average	2011.56	1931.64	79.92
Minimum	640.03	595.25	7.93
Maximum	4627.72	4389.90	410.49
Standard Deviation	983.18	903.29	119.40
Median	1797.37	1789.18	27.88
Variance	966633.37	815927.82	14256.48
Number of Outliers	2	2	2

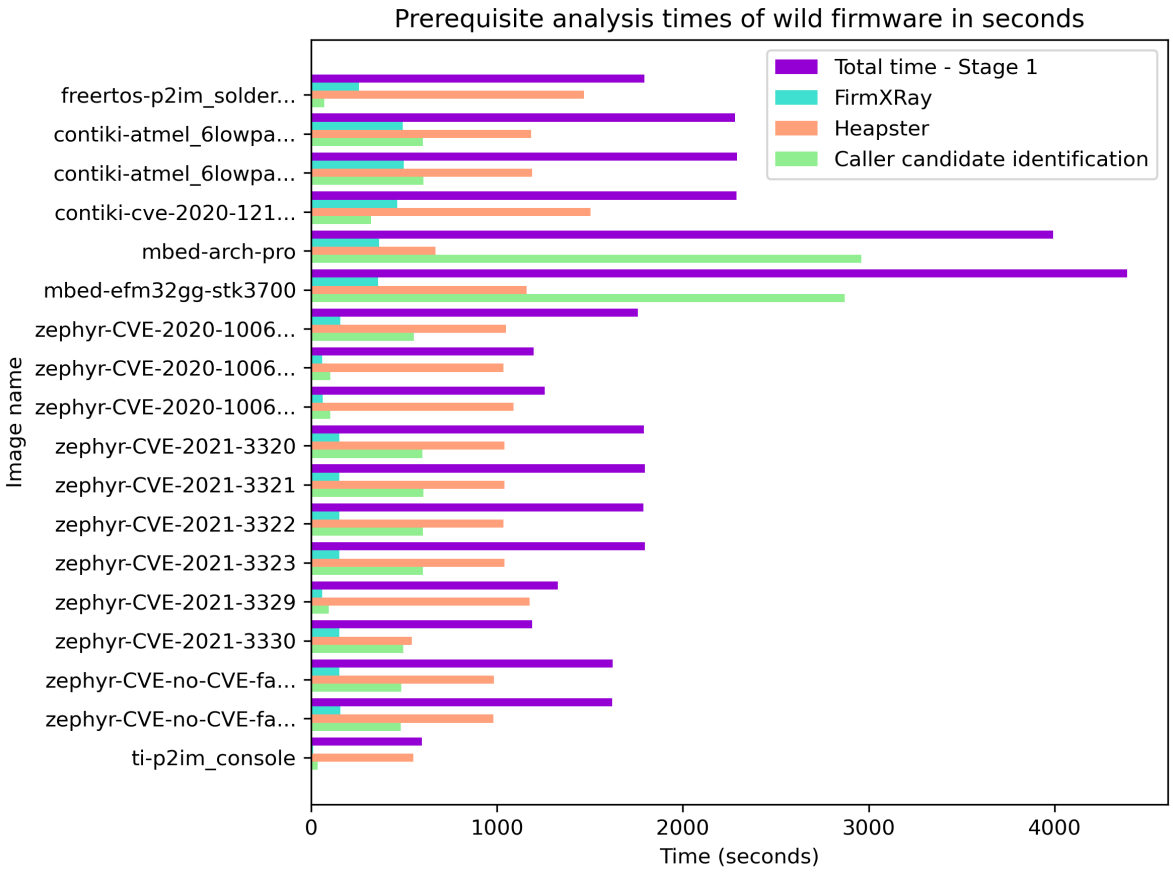

Figure 5.7: This figure shows the execution times of each step in the first stage of analysis and how these compare to the total execution time.

Table 5.11: Statistics for the peformance results of applying stage 1 analysis to wild firmware

	Stage 1 performance	**FirmXRay performance**	**Heapster performance**	**Caller candidate identification**
Statistic	Statistical values			
Average	1931.64	214.00	1040.62	677.02
Minimum	595.25	11.00	542.19	34.69
Maximum	4389.90	497.41	1503.56	2959.56
Standard Deviation	903.29	150.50	248.21	819.18
Median	1789.18	152.16	1040.92	523.84
Variance	815927.82	22649.75	61609.48	671060.87
Number of Outliers	2	0	1	2

Lastly, we want to gain more insight into the performance times of the second and third stages of analysis when our framework is applied to each image in the wild firmware dataset. The results of this analysis are shown in figure 5.8. During the performance analysis of applying stage 2 and stage 3 to the ground truth dataset, we found that generally, stage 3 has a near negligible share of the performance overhead. We now analyze the execution times shown in figure 5.8.

Again, we find that the third stage of analysis has a negligible impact on overall performance in all but three cases. We want to understand why analysis takes longer in the three Zephyr-based firmware images, as all three are a statistical outlier, as shown in table 5.12. First of all, we note that these images are all compiled with what is most likely the same application: an application that sends advertisements to a device over a Bluetooth connection. This has been verified by means of manual analysis. We were able to determine that when PinDown is analyzing the main application during the third analysis stage, we encounter a function that introduces several finite loops. Due to how we aim to uncover information about these functions by means of register and memory analysis, we can assume that iterating over these loops introduces the observed performance overhead. This confirms our earlier observation that

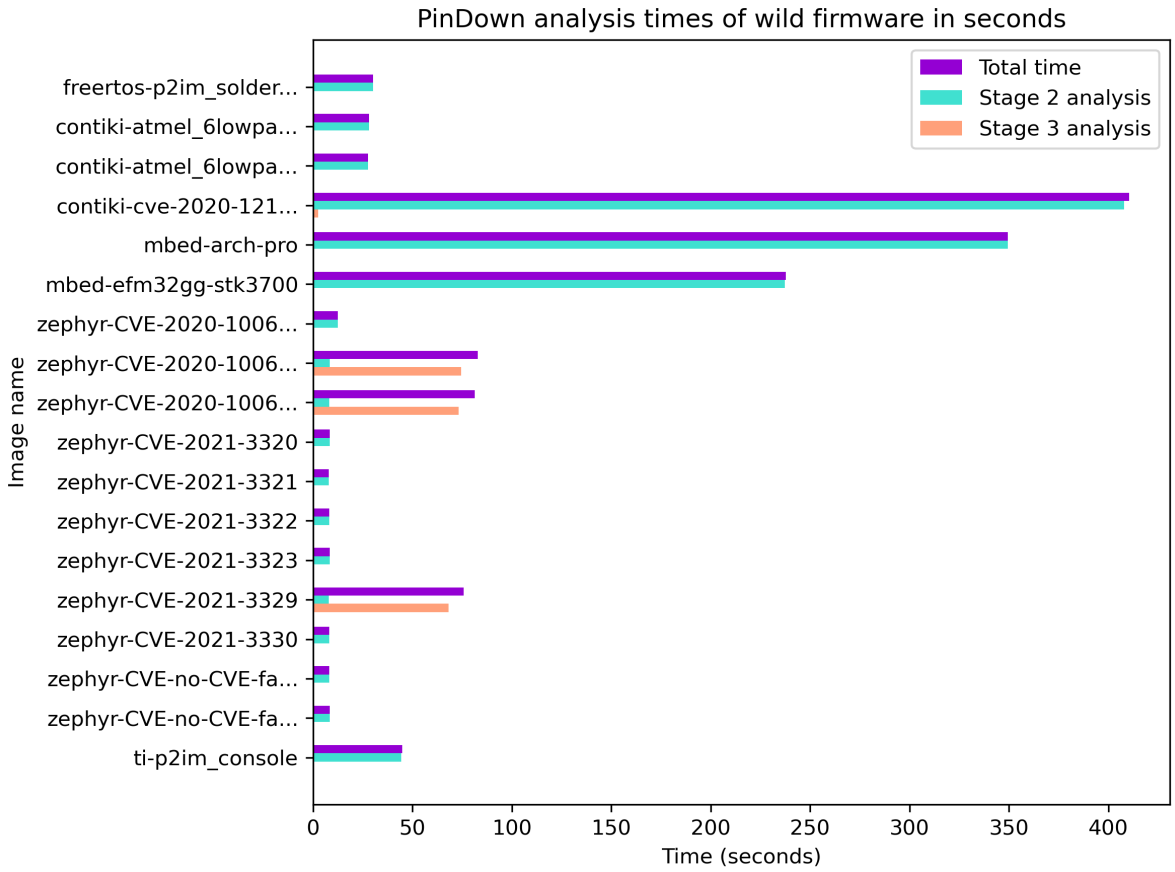

Figure 5.8: An overview of the performance results of applying stage 2 and stage 3 analysis to each image in the wild firmware dataset. The total time for each image is equal to the sum of stage 2 and stage 3 times.

because of how we aim to deal with potential infinite loops, the performance is partially based on the implementation of user applications.

Table 5.12: Statistics of the performance results acquired from executing stage 1 and stage 2 on wild firmware

	Total execution time	**Stage 2 analysis**	**Stage 3 analysis**
Statistic	Statistical values		
Average	79.92	67.72	12.20
Minimum	7.93	7.82	0.00
Maximum	410.49	407.98	74.48
Standard Deviation	119.40	121.92	26.73
Median	27.88	8.42	0.03
Variance	14256.48	14864.40	714.63
Number of Outliers	2	2	3

6

Conclusion, Limitations And Future Work

In section 6.1 of this chapter, we discuss the results of the experimental validation and provide the answers to the initial research questions. Furthermore, this section discusses the limitations of the work. Finally, section 6.2 provides a short summary of the work and discusses potential future work.

6.1. Discussion

Based on the results of the experimental validation, it becomes clear that our current approach and implementation for identifying user application code is effective when concerned with binary firmware images based on RIOT, mBed OS, Zephyr, NuttX, and LittleKernel. The results show that we can correctly identify functions that implement thread creation that are part of the corresponding RTOS based on their relation to heap modifying functions. Furthermore, the results show that functions that implement thread creation can be leveraged to identify functional components and user application code. We also show that our alternative method of function discovery by means of analyzing compiled structures is effective at identifying such functions when reaching definitions analysis is insufficient. When inspecting the results of the ground-truth dataset, we see that our framework is only ineffective at identifying application code in one group of firmware. This can be attributed to pointers referencing addresses from the virtual address table, which are not statically accessible. Inspecting the results from the wild firmware dataset, we find that incorrectly identified functions, omitted functionality, and a lack of reliance on RTOS components are the main reasons why our framework failed to identify application code. We also find that the performance of our approach and implementation is impacted by the type of application that is compiled for the system. This is because we may end up analyzing user application code when we need to discern whether or not we are dealing with a functional component, an initialization mechanism, or actual user application code. This application code may initialize infinite loops which we cannot statically detect. Due to the sheer variety of applications, this introduces performance overhead. However, the overhead introduced by our implementation is minimal compared to the overall overhead introduced by the first stage of analysis within both the ground truth and wild firmware datasets. A great portion of the overhead can be attributed to how Heapster uncovers HML components as it does so by simulating existing functions. Furthermore, the process of applying reaching definitions analysis in order to uncover potential functions that create threads is responsible for the majority of the performance overhead. We also find that the methods we implement to identify potential candidate functions in the event stage 1 analysis are unsuccessful, contributing to an increase in execution time given the exhaustive nature of the implementations. The implemented methods also increased the false positive rate of the identified candidates, which was expected.
Addressing which types of initialization mechanisms are prevalent in RTOS-based firmware (**SQ1**), we find that thread creation is the prevalent method of initiating user application code either by a functional component or directly in RTOS-based firmware. We also find that we can identify user application code and functional components by analyzing register values, which are passed as arguments when such a function that implements thread creation is called. In addition, we have found that these addresses are stored in data structures within memory regions of the image, and these data structures require different detection mechanisms, addressing how we can leverage these mechanisms to enable appli-

cation code discovery (**SQ2**). Lastly, based on our results, we find that our framework is applicable to a variety of different RTOS-based systems but that we are limited by the current technology that enables accurate function identification within stripped firmware. Furthermore, we find that when the RTOS library functions for systems have been modified, or that thread creation components have been omitted, our analysis is unsuccessful in addressing the extent to which our approach and implementation are applicable (**SQ3**).

We also gain a better understanding of the limitations of our approach. We find that it is important to devise a method of identifying data structures within binary firmware images and the entries within these data structures. Experiments concerned with identifying function addresses in Contiki-NG-based firmware halt further analysis as we cannot identify the valid function addresses stored in a list. This is because the construction of that list was very different from what we had encountered during the code analysis of RTOS-based firmware. When we can more accurately identify such data structures, we should be able to better identify user applications within such binary firmware images. Another limitation we identify is that reaching definitions analysis cannot always reach a candidate function that implements thread creation. This is because the assumption that such functions depend on HML functions to accommodate user applications was partially incorrect. We show that leveraging HML functions by applying reaching definitions analysis is effective when a function that implements thread creation does depend on HML components. Furthermore, we find our solution to be dependent on the presence of thread creation as provided by an RTOS framework. This also implies that our approach is limited to analyzing binary firmware images in which the framework and its components remain unaltered. Furthermore, our research was limited by the lack of availability of device firmware that fit our scope and the fact that many open-source RTOS frameworks have no documentation, rely on devices to create firmware, or only provide support in Chinese, which we could not translate. All of which shrunk the potential dataset we could analyze. In order to combat this lacking availability of device firmware research and RTOS frameworks, we have tried to increase the size of the dataset of wild firmware by decompiling bit code files into `.bin` files but this resulted in a missing function hook, preventing the creation of call- and control-flow graphs[1].

6.2. Conclusion

In this work, we have presented PinDown, an automated analysis framework that identifies application code within RTOS-based firmware without requiring partial system knowledge. We have provided an analysis of different RTOS frameworks as well as the initialization characteristics and mechanisms within firmware based on these frameworks. Using this analysis, we have constructed an approach that identifies application codes within firmware based on six different RTOSs. This is achieved by leveraging heap modifying functions to identify RTOS components that enable the discovery of application code. As our approach is generalizable, it can be applied to any RTOS-based firmware, but its success depends on unmodified RTOS library functions as well as the correct identification of functions by the disassembler. We have performed experimental validation by means of analyzing a custom set of firmware images as well as wild firmware images and found that PinDown was able to reduce the number of candidate functions in a binary by 99.18% on average, sporting a 50.26% false positive rate. PinDown also addressed several challenges inherent to the field of firmware analysis and was able to mitigate the potential effects these would exert on the effectiveness.

There are several options that future work could investigate to improve PinDown or build upon the existing framework. An alternative method of identifying RTOS components would enable a more accurate and efficient analysis as reaching definitions analysis depends on correct function identification. Furthermore, the work can be improved by being able to identify structures within firmware images that store data such as function addresses and index separators. An extension to this work could focus on the automated analysis of the application code identified with PinDown in search of vulnerabilities. Lastly, as PinDown identifies RTOS components to discover application code, future work could utilize this approach in order to determine the type of framework present in the firmware based on the identified functions.

[1] bitcode database: `https://github.com/RTOSExploration/lctes2023-artifact/tree/main`

Bibliography

[1] L. S. Vailshery. *Number of Internet of Things (IoT) connections worldwide from 2022 to 2023, with forecasts from 2024 to 2033*. 2024.

[2] Simon Duque Anton et al. “Two decades of SCADA exploitation: A brief history”. In: *2017 IEEE Conference on Application, Information and Network Security (AINS)*. IEEE. 2017, pp. 98–104.

[3] Ana Kovacevic and Dragana Nikolic. “Cyber attacks on critical infrastructure: Review and challenges”. In: *Handbook of Research on Digital Crime, Cyberspace Security, and Information Assurance* (2015), pp. 1–18.

[4] Amir Djenna, Saad Harous, and Djamel Eddine Saidouni. “Internet of things meet internet of threats: New concern cyber security issues of critical cyber infrastructure”. In: *Applied Sciences* 11.10 (2021), p. 4580.

[5] Ibrahim Nadir, Haroon Mahmood, and Ghalib Asadullah. “A taxonomy of IoT firmware security and principal firmware analysis techniques”. In: *International Journal of Critical Infrastructure Protection* (2022), p. 100552.

[6] Dimitrios Pliatsios et al. “A survey on SCADA systems: secure protocols, incidents, threats and tactics”. In: *IEEE Communications Surveys & Tutorials* 22.3 (2020), pp. 1942–1976.

[7] Hugo Riggs et al. “Impact, vulnerabilities, and mitigation strategies for cyber-secure critical infrastructure”. In: *Sensors* 23.8 (2023), p. 4060.

[8] Frank Ebbers. “A large-scale analysis of iot firmware version distribution in the wild”. In: *IEEE Transactions on Software Engineering* 49.2 (2022), pp. 816–830.

[9] Abdullah Qasem et al. “Automatic vulnerability detection in embedded devices and firmware: Survey and layered taxonomies”. In: *ACM Computing Surveys (CSUR)* 54.2 (2021), pp. 1–42.

[10] Marius Muench et al. “What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices.” In: *NDSS*. 2018.

[11] Nilo 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.

[12] Sebastian Eschweiler, Khaled Yakdan, Elmar Gerhards-Padilla, et al. “Discovre: Efficient cross-architecture identification of bugs in binary code.” In: *Ndss*. Vol. 52. 2016, pp. 58–79.

[13] Haohuang Wen, Zhiqiang Lin, and Yinqian 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.

[14] Fabio Gritti et al. “Heapster: Analyzing the security of dynamic allocators for monolithic firmware images”. In: *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2022, pp. 1082–1099.

[15] Pallavi Sivakumaran and Jorge Blasco. “argXtract: Deriving IoT security configurations via automated static analysis of stripped arm cortex-m binaries”. In: *Proceedings of the 37th Annual Computer Security Applications Conference*. 2021, pp. 861–876.

[16] Mingyi Huang and Chengyu Song. “ARMPatch: A binary patching framework for ARM-based IoT devices”. In: *Journal of Web Engineering* 20.6 (2021), pp. 1829–1852.

[17] Jiongyi Chen et al. “IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing.” In: *NDSS*. 2018.

[18] Thoms Ball. “The concept of dynamic analysis”. In: *ACM SIGSOFT Software Engineering Notes* 24.6 (1999), pp. 216–234.

[19] Michael D Ernst. “Static and dynamic analysis: Synergy and duality”. In: *WODA 2003: ICSE Workshop on Dynamic Analysis*. 2003, pp. 24–27.

[20] Ryan Roemer et al. “Return-oriented programming: Systems, languages, and applications”. In: *ACM Transactions on Information and System Security (TISSEC)* 15.1 (2012), pp. 1–34.

[21] Victor Alessandrini. *Shared memory application programming: Concepts and strategies in Multi-core application programming*. Morgan Kaufmann, 2015.

[22] Jiunn-Yeu Chen et al. “Effective code discovery for ARM/Thumb mixed ISA binaries in a static binary translator”. In: *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. IEEE. 2013, pp. 1–10.

[23] Jiunn-Yeu Chen et al. “On static binary translation of arm/thumb mixed isa binaries”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 16.3 (2017), pp. 1–25.

[24] Paolo Tonella et al. “Variable-precision reaching definitions analysis”. In: *Journal of Software Maintenance: Research and Practice* 11.2 (1999), pp. 117–142.

[25] Sandro Pinto and Cesare Garlati. “Secure IoT Firmware For Cortex-M Processors”. In: ().

[26] Pietro De Nicolao et al. “ELISA: ELiciting ISA of raw binaries for fine-grained code and data separation”. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2018, pp. 351–371.

[27] Richard Wartell et al. “Differentiating code from data in x86 binaries”. In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer. 2011, pp. 522–536.

[28] Ruijin Zhu et al. “Determining image base of firmware files for ARM devices”. In: *IEICE TRANSACTIONS on Information and Systems* 99.2 (2016), pp. 351–359.

[29] Eric Gustafson et al. “Shimware: Toward Practical Security Retrofitting for Monolithic Firmware Images”. In: *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. 2023, pp. 32–45.

[30] Jintao Huang et al. “TaiE: Function Identification for Monolithic Firmware”. In: *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*. 2024, pp. 403–414.

[31] Xiaokang Yin et al. “Function recognition in stripped binary of embedded devices”. In: *IEEE Access* 6 (2018), pp. 75682–75694.

[32] Andreas Ibing and Alexandra Mai. “A fixed-point algorithm for automated static detection of infinite loops”. In: *2015 IEEE 16th International Symposium on High Assurance Systems Engineering*. IEEE. 2015, pp. 44–51.