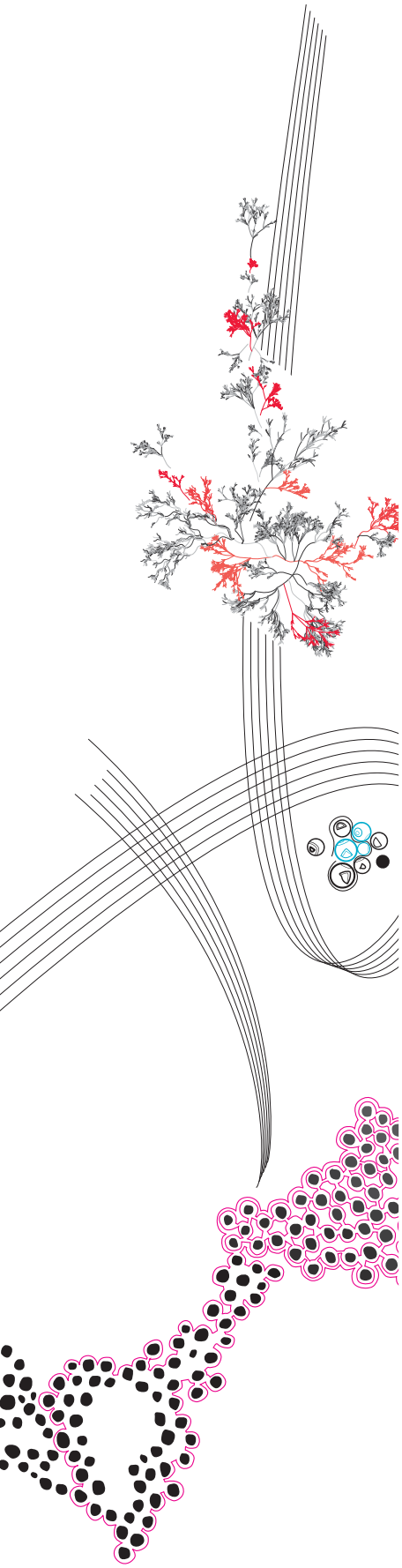


MSc Computer Science
Final Project

XtractIO: Statically locating MMIO addresses in non-Linux firmware

Mitchel Scholtens

Supervisor: Jorik van Nielen
Andrea Continella

July, 2025

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Abstract

As embedded devices become increasingly integrated into our everyday lives, so does the risk of malicious actors compromising our security. To secure these devices, various analyses, including static and fuzz-testing (fuzzing), are developed to identify vulnerabilities before they are exploited. One key area of research is Memory Mapped Input-Output (MMIO), as these enable untrusted input from peripherals (such as Bluetooth and Ethernet) to reach the CPU.

A common approach researchers use to extract MMIO addresses is to manually look them up in datasheets and other types of documentation. This is possible when analysing minimal amounts of firmware. However, this will be challenging to do when we need to examine them in bulk. Additionally, when the microcontroller (MCU) is unknown, we are also unable to locate its datasheet, making MMIO address lookups impossible. To facilitate analysis in these situations, we developed XtractIO, which automatically locates MMIO addresses in non-Linux firmware. Our approach uses patterns to locate MMIO operations, after which it extracts the referenced addresses.

We evaluated our approach against a set of 40 firmware images consisting of ARM Cortex-M and Xtensa architectures. XtractIO performs, on average, 30 per cent better compared to a naive approach. However, even though XtractIO outperforms a naive approach, it still has an average F1-score of 60 to 70 per cent, indicating that it still produces both false positives and negatives. These results demonstrate that it is possible to locate MMIO addresses automatically, but further research is necessary to make the results reliable.

1 Introduction

Nowadays, a wide variety of embedded devices are connected to the Internet, with devices ranging from smart speakers and smart doorbells to adaptive cruise controls found in cars. The market for embedded devices was valued at 94.77 billion dollars in 2022 [1]. This market increased to 100.04 billion dollars in 2023 and is projected to reach a value of 171.86 billion dollars in 2030. With the popularity of embedded devices, there has also been an increase in attack frequency [2], where the exploited vulnerabilities can range from buffer overflows [3] to default credentials [4] and from improper authentication to injection [5].

To safeguard embedded devices, it is essential to identify vulnerabilities before they are exploited. Two popular research directions are static analysis and fuzz-testing (fuzzing). Static analysis allows us to detect vulnerabilities without executing the code, while a fuzzer automatically detects vulnerabilities and defects by injecting invalid, malformed, or unexpected inputs [6]. An advantage of these approaches is that they are both automated, which saves a considerable amount of time compared with methods like manual code review.

Fuzzing is often done using re-hosting [7–12], which allows embedded firmware to run in an emulated environment with more resources than would be available on the device itself. This enables a higher throughput, resulting in improved fuzzing performance. Another advantage is that we only need the firmware for testing and not the hardware. For re-hosting, three main interactions must be emulated: Memory Mapped Input-Output (MMIO), Direct Memory Access (DMA) and interrupts [7]. The manufacturers have fortunately described many of the necessary addresses for these interactions in datasheets.

MMIO addresses are one of the interactions that need to be emulated for re-hosting. Through MMIO addresses, peripherals communicate and exchange data with the CPU via read and write instructions [13]. These instructions appear identical to regular read and write instructions, such as those used for RAM and flash [14]. Combining this with the fact that MMIO addresses also share the same memory address space as RAM and flash, distinguishing them from one another can be challenging [15].

Since peripherals are external hardware devices, data obtained through MMIO addresses should be considered untrusted by the system. Malicious actors can manipulate this data to change the firmware's behaviour, potentially causing unwanted and harmful actions. An example of manipulating the input is by altering the altitude of a drone by modifying the values of the Barometer [16]. As a result, the drone performs differently than intended. Another example is that by using a Bluetooth peripheral, a malicious actor can remotely alter the current or voltage of a power supply, causing damage to the unit and the

connected device [17]. To prevent untrusted input from performing unwanted and malicious behaviour, we want to locate the MMIO addresses. Once located, these can be used for automated analysis, such as fuzzing or static analysis.

The majority of works locate MMIO addresses by manually looking them up from datasheets for their analysis [7–12, 18–23]. This method is possible when analysing one firmware at a time. However, relying on datasheets introduces manual work, which prevents scaling the analysis to a large batch of firmware. Another limitation is that some works extract firmware from over-the-air updates [24]. Although they identify the architecture through various methods, they do not determine the specific microcontroller (MCU) for which the firmware was built. Without this information, we cannot look up the MMIO addresses from the corresponding datasheet.

We aim to eliminate these limitations by automatically locating MMIO addresses solely from the firmware and its architecture. To do this, we first identify patterns in firmware that indicate MMIO addresses. After this, we develop static analysis methods based on those patterns to locate MMIO addresses. Finally, we incorporate these methods into a pipeline and evaluate the performance on ARM Cortex-M and Xtensa firmware images.

Our evaluation shows that the patterns we identified locate numerous MMIO addresses. Having a score between 60 and 70 per cent F1-score on average. Which is higher than looking at it naively, scoring only between 30 and 40 per cent on average. However, despite the improvement compared to the naive approach, we still introduce false positives and negatives because our patterns do not exclusively locate MMIO addresses and fail to locate all MMIO addresses. Ultimately, our results show that automatically locating MMIO addresses is possible, but further research is necessary to make it more reliable.

2 Background

This research focuses on identifying MMIO addresses within non-Linux firmware. First, we briefly explain the properties of non-Linux firmware and how it differs from other firmware. Then, we introduce MMIO, emulation, intermediate representation language and firmware region identification. Finally, we explain different kinds of firmware analysis where MMIO addresses can be used as input.

2.1 Firmware types

Firmware is the software running on embedded devices, containing all the instructions to start up, transmit data to other devices, and interact with the physical world through peripherals [25]. Firmware for embedded devices falls into three categories: Type-1, Type-2 and Type-3 [26].

- **Type-1 firmware** consists of a general-purpose Operating System (OS) often combined with a lightweight user-space environment and security modules like the Memory Management Unit (MMU). It provides the most functionality, security and flexibility. Type-1 firmware typically includes debug symbols. These symbols contain metadata such as the location of variables and functions, descriptions of code structures and other related information [27]. Using the symbols makes it rather easy to extract the architecture and base address of the firmware.

- **Type-2 firmware** is tailored for lower-power devices. It often has a specific-purpose OS with a logical separation between the application and the kernel, making it more functional than Type-3 firmware. However, Type-2 (and Type-3) firmware is generally more prone to errors than Type-1 firmware because it lacks modules like the MMU, which can lead to undetected memory corruption bugs. Type-2 firmware also lacks debugging symbols, making it more challenging to extract the architecture and base address. This Type is often written in C and compiled into a single binary. In this binary, the data and instruction are interwoven with each other.

- **Type-3 firmware** has the least amount of resources compared with the other categories. It operates without an OS, relying on a single-loop structure. This loop is only broken when peripherals that receive external information trigger an interrupt. Like Type-2, it is typically written in C, combines data and instructions into one binary, and lacks

debugging symbols [7]. Its architecture and base address are, like Type-2, also unknown.

What distinguishes Type-1 firmware from Type-2 and Type-3 is that Type-1 has a filesystem with a known structure, whereas Type-2 and Type-3 combine code and data into a single executable that lacks debugging symbols. Identifying the architecture and base address without these symbols requires additional effort, making the security analysis more challenging. If the base address and architecture are unknown, then it would be hard to decompile the code. Picking the wrong architecture will lead to unreadable assembly, and having the incorrect base address will lead to absolute pointers referencing incorrect addresses [28].

This research will focus on Type 2 and Type 3 firmware, which we call "non-Linux firmware".

2.2 MMIO

MMIO is a mechanism by which peripheral registers are mapped to the CPU's memory address space as fixed memory addresses [29]. As those registers originate from external hardware, MMIO presents a source of untrusted input for the CPU. Using MMIO, the CPU can interact with peripherals such as Bluetooth, Display, and Ethernet using ordinary read/write instructions [13]. The MMIO memory range of a peripheral consists of multiple segments, each segment representing a different MMIO register that serves a different purpose. A few examples of these registers are status, data and configuration registers. The status register is used by the peripheral to communicate its state to the CPU, the configuration register configures the peripheral, and the data register sends or receives data from another source. The values of these registers can change frequently based on what they do [7, 8, 10, 17].

Similar to MMIO, DMA and Port-Mapped Input/Output (PMIO) can also access peripherals. However, this work focuses solely on MMIO for two reasons. Firstly, we are interested in the communication between peripherals and the CPU. Secondly, the memory range used by MMIO is the same as for RAM and flash [15]. Where the read-/write operations and the instructions that perform them look the same [14], making them hard to distinguish. Even though this second reason also applies to DMA, what makes it different is that DMA has a specific hardware controller that communicates with the peripherals, circumventing the need for the CPU [7, 30]. While PMIO are mapped to an entirely different address space [31]. A visual comparison between MMIO, PMIO and DMA can be seen in fig. 2.

To ensure accurate identification of MMIO registers, manufacturers often release datasheets of their MCU, including a detailed memory map of the MMIO region. You can find an example of this in fig. 1, which shows the memory map for the ARM STM32F103x MCU. Here, you can see that the entire peripheral segment spans from `0x40000000` to `0x60000000` with different ranges allocated for different peripherals [32].

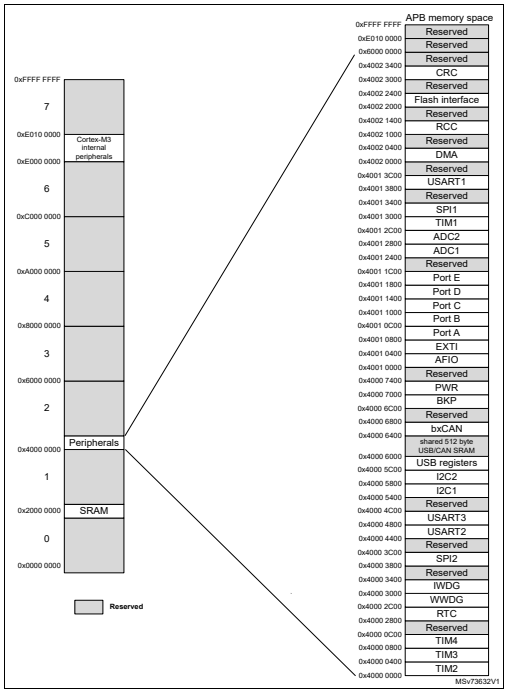


FIGURE 1: The MMIO address range with peripherals for the STM32F103x MCU. The range spans from address `0x40000000` to `0x60000000` and contains multiple different peripherals and reserved spaces [32].

2.3 Emulation

Emulation is an approach to imitate the behaviour of software or firmware [33]. It can be used to connect devices, run software on different hardware or provide the software with more resources. The last two cases are often combined for embedded firmware analysis, which is called re-hosting.

For re-hosting to be possible, a few interactions must be emulated: MMIO, DMA and inter-

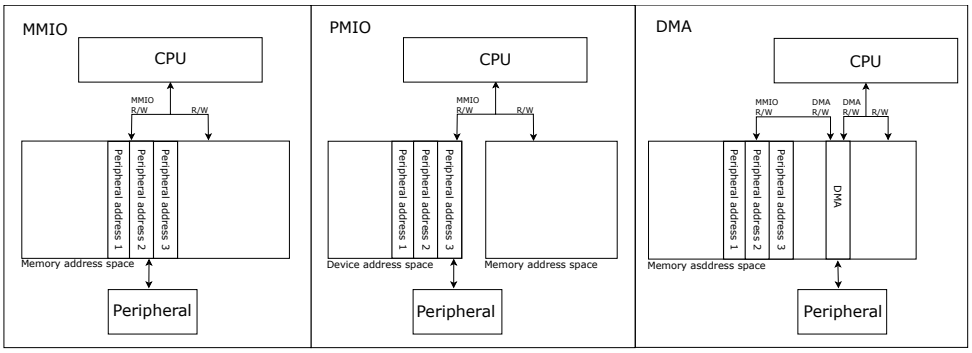


FIGURE 2: The difference between MMIO, PMIO and DMA from the firmware's perspective. The MMIO on the left has only one address space. The PMIO in the middle has two address spaces. One for the main memory and one for the device. DMA on the right also has one address space. However, unlike MMIO (and PMIO), DMA uses a dedicated mechanism for communication between peripheral hardware and peripheral addresses without CPU intervention.

rupts [7]. The addresses for these interactions are commonly documented in datasheets by the MCU manufacturer. As previously mentioned, MMIO enables the firmware to communicate with the peripherals, while DMA facilitates this communication too but without CPU intervention. Interrupts differ from DMA and MMIO, as they allow the CPU to pause current processes to facilitate higher-priority tasks [34]. After this task is completed, the CPU returns to normal execution by continuing the paused process. Once emulated, re-hosting allows for dynamic analysis, where tools such as fuzzing can be used to automate test cases.

2.4 Intermediate Representation

An intermediate representation (IR) is a representation of a program that exists between the high-level program language and low-level assembly code [35, 36]. IR has three levels: High, Medium and Low [37]. High-level IR resembles the high-level program language, while low-level IR resembles the underlying assembly. Finally, middle-level IR is situated in the middle, trying to be independent of both the high-level program language and the low-level assembly code. IR can be used to simplify analysis. This is because different languages have different syntax for the same operation. In listing 1 you can see an example of these different syntaxes in assembly. By using IR, we can translate this to one kind of instruction as seen in listing 2. Here, the only difference is that the register name differs per language, while the whole structure of the instruction itself stays the same.

```
1 ldr       r2, [r3,#0x28]
2 l32i      a2, a3,#0x28
3 lw        a2, 0x28(a3)
```

LISTING 1: LDR instruction of ARM Cortex-M, Xtensa and RISC-V respectively.

```
1 unique[98000:4] = $3 + 0x28
2 $2 = *[ram]unique[98000:4]
```

LISTING 2: LDR instruction using the low level IR pypcode ($ is the placeholder for the register letter).

2.5 Firmware region identification

Identifying MMIO regions in firmware and automatically obtaining information about them requires a solid understanding of firmware analysis techniques, which can be split into static and dynamic approaches.

2.5.1 Static analysis

Static analysis analyses the firmware without requiring execution, eliminating the need to know the datasheet entirely beforehand. Different static analysis techniques are disassembling, decompilation and symbolic execution. With disassembling, you break a binary down to its assembly language,

which is a human-readable language [38]. Decompilation is a technique that reverts a binary to high-level code without altering its functionality [39]. Finally, symbolic execution is used to execute a program abstractly. Instead of using concrete values, it simulates multiple executions at the same time using different inputs. Symbolic execution focuses on parts of the code where the inputs follow the same execution. By using this technique, symbolic execution can identify and return constraints representing the output for those inputs [40, 41].

Although static analysis does not need prior knowledge of the memory map, it does introduce some challenges. To be precise, firmware images often rely on absolute pointers, which are pointers that are fixed to a certain memory address. A challenge that can be introduced because of this is a mismatch between normal execution and static analysis. This is because decompiler tools use a standard base address from which they start loading the firmware if no information is provided. The absolute pointers that are present in the firmware are only usable if the right base address is used, meaning if this is not the case it can cause issues. While contributions are made to resolve this problem [28], it still relies on having enough absolute pointers available to build off of, which is not always the case. This challenge is not limited to static analysis because, in dynamic analysis, the environment needs to know what the base address is to load in the firmware correctly.

2.5.2 Dynamic analysis

In contrast to static analysis, dynamic analysis allows us to analyse non-Linux firmware during run-time. For dynamic analysis of the MMIO addresses, prior information about the different memory segments is necessary. If we know these segments, we can use techniques such as information interception, where we, for example, can define a boundary using a segment and use it to intercept addresses found within [7]. Another approach to dynamic analysis is by defining all segments in the firmware, leaving a single one out. When the firmware runs this way, crashes or undefined behaviours can occur when specific addresses in this undefined segment are accessed, which can then be extracted [42]. This dynamic analysis technique is our educated guess based on the paper from Gui e.a. However, because it is not explicitly stated, we cannot confirm this. Dynamic analysis introduces some challenges, as we need to know a significant amount of information beforehand so that the emulator can work correctly. Gathering this information will be problematic if the MCU of the firmware is unknown, because we cannot obtain this information from, for example, the datasheet.

2.6 Vulnerability analysis

2.6.1 Fuzzing

Fuzzing is an automated technique that can detect vulnerabilities by feeding the target with manipulated inputs and watching it till an exception occurs. These inputs are mutated based on the feedback received from the fuzzer during program execution. It is often used for testing the blockchain, OS kernels [43] and firmware. Fuzzing has a high throughput resulting in many cases being tested in a short time.

A popular fuzzing approach is coverage-guided fuzzing, where the fuzzer uses coverage feedback from past executions to find new inputs [10, 43–45]. The fuzzer adds instrumentation tools to the target, which report back different kinds of coverage information, like bitmaps, which are used to track branch execution. By using these tools, the fuzzer can, for example, keep track of new edge cases. The fuzzer can then save this information and use it to mutate a new input which is similar to the original. Using this approach, a fuzzer can steer towards more interesting use cases than a completely random approach.

Where regular software fuzzing focuses on discovering vulnerabilities in software applications [46], firmware fuzzing's primary focus is on MMIO addresses. These addresses facilitate communication exchange with peripherals in the system [7]. For this, it is necessary to know beforehand where these addresses reside in the memory. However, in many cases, finding MMIO addresses is challenging and often relies on documentation. How the documentation is used differs from case to case. In one approach, they use the whole documentation as input [22], while in another approach, they use it as a boundary for their heuristic search [7].

Currently, most research uses the entire MMIO segment as input for fuzzing [10, 19]. However, in this segment multiple peripherals exist, each with its own distinct MMIO addresses (as shown

in fig. 1). Knowing these individual MMIO addresses can be valuable because it allows us to develop fuzzing techniques that can handle specific inputs from these different kinds of addresses. This approach can then improve the efficiency and accuracy of testing.

2.6.2 Static analysis

Static analysis can identify bugs and vulnerabilities without executing the firmware. By analysing assembly instructions directly, it can discover logical bugs like race conditions [47, 48], MMIO-related bugs [17, 23, 28], and more.

Symbolic execution is a static analysis technique that can be used for this. The technique uses predefined entry points within the firmware as a starting point and traverses all execution paths until specific conditions are met [17]. These conditions can be, for example, predefined rules or constraints. Another input for symbolic execution are MMIO addresses [12]. These addresses can be used as symbolic data to identify MMIO models. During execution, the code can return different consecutive values from the same MMIO address. By using these different values, it is possible to create constraints for an MMIO address, which can then be used for an MMIO model. These models can, among other things, be used as input for emulating firmware.

A second technique that can be used for statically analysing the firmware is a Control Flow Graph (CFG) [28]. A CFG is a graphical representation of the flow within firmware [49]. It represents the possible paths of execution by showing basic blocks as nodes and the connection between them as edges. By using a CFG, it is possible to show different kinds of transitions, such as if-else statements, (do-)while loops and for loops. By analysing the CFG, it is possible to track the execution flow. This can be done in a forward and backward way. Backward traversal allows the identification of instructions that are responsible for calculating values of relevance, such as configuration values. Forward traversal shows how the point of interest affects later operations. An example of how a CFG can be used is for calculating MMIO configuration values [28]. These values can then be analysed using external documentation or tools to identify vulnerabilities.

A static analysis technique that uses the CFG is data flow analysis [50]. This technique analyses how the data flows through a program. It tracks variables and expressions as they are being used. Data flow analysis uses the CFG to propagate its information [51].

Finally, it is also possible to use custom techniques to extract information from the firmware. In these cases, custom patterns or heuristics are employed to identify, for example, bugs or vulnerabilities. A possible input for these kinds of analyses are MMIO addresses. Current research shows that MMIO addresses can be used to identify possible locations for attacker-controlled input [17] and to define the boundary of an interaction between the firmware and peripherals [23]. This boundary can then help identify, for example, concurrency bugs.

2.7 Contributions and goals

We assume for our research that we have a binary firmware image with prior knowledge of its underlying architecture (ARM Cortex-M or Xtensa). Using these as input, we want to automatically locate the MMIO addresses from the firmware image so that they can be used for further static and dynamic analysis. To achieve the automatic extraction of MMIO addresses, we did the following:

- The identification of firmware MMIO patterns used for locating MMIO addresses.
- The design of a firmware analysis pipeline that can detect MMIO addresses through pattern analysis.
- The evaluation of the pipeline's performance on a ground truth collection consisting of ARM Cortex-M and Xtensa firmware samples.

We address the stated contributions and goals in the remainder of this thesis.

3 Identification of MMIO patterns

To identify MMIO address patterns in firmware, we manually analyse ARM Cortex-M and Xtensa firmware samples to build an intuition on how firmware uses MMIO addresses. We distil patterns that occur for both architectures to make our approach versatile.

Our analysis results in five patterns. In the remainder of this section, we describe the different patterns that we identified with disassembly examples from real-world firmware. For the IR of the pattern examples, we refer to appendix A.

Load, Modify, Store (LMS): The first pattern we identified is referred to as LMS. When data communication occurs between the peripheral and the CPU, the peripheral's state changes. Think, for example, of being busy when data is being transferred or free when it is not. We observe that this is reflected in the firmware, where data is first loaded from an MMIO address. Following this, a bitwise operation is performed (either AND or OR). Finally, the result of this will be stored at the MMIO address from which it was previously loaded. This results in changing a set of bits within the peripheral or entirely clearing the bits. The pattern can be seen in listing 3.

```
129e8  5b 6c   ldr   r3,[r3,#0x44] => DAT_40023844
129ea  43 f0   orr   r3, r3,#0x10
       10 03
129ee  53 64   str   r3,[r2,#0x44] => DAT_40023844
```

LISTING 3: The load, modify, store pattern in ARM Cortex-M assembly.

Function Related Instructions (FRI): The second pattern we identified is referred to as FRI. When an LMS pattern occurs in a function, the firmware tends to have the remaining load and store instructions within that function also interact with peripherals. The reason for this is that you want to have the least amount of processing between MMIO instructions. If more processing is done between instructions, then there is a higher chance of an interrupt being triggered. This is because an interrupt can be triggered at any time [52]. An interrupt can lead to a delay in, for example, data exchange, which can halt other processes down the line. We observe the FRI pattern in the assembly as multiple load and store instructions communicating with peripherals outside of the LMS pattern within a function, as illustrated in listing 4. The amount of LMS patterns occurring in a function can vary. We identified that an increase in LMS patterns within a function often indicates that the remaining load and store instructions are also more likely

```
6bce  9a 69   ldr   r2,[r3,#0x18] => DAT_40021018
6bd0  42 f0   orr   r2, r2, #0x4
      04 02
6bd4  9a 61   str   r2,[r3,#0x18] => DAT_40021018
6bd6  9b 69   ldr   r3,[r3,#0x18] => DAT_40021018
...
6be8  9a 69   ldr   r2,[r3 #0x18] => DAT_40021018
6bea  42 f0   orr   r2, r2, #0x8
      08 02
6bee  9a 61   str   r2,[r3,#0x18] => DAT_40021018
6bf0  9b 69   ldr   r3,[r3,#0x18] => DAT_40021018
...
6c00  9a 69   ldr   r2,[r3,#0x18] => DAT_40021018
6c02  42 f0   orr   r2, r2, #0x10
      10 02
6c06  9a 61   str   r2,[r3,#0x18] => DAT_40021018
6c08  9b 69   ldr   r3,[r3,#0x18] => DAT_40021018
...
6c18  9a 69  ldr   r2,[r3,#0x18] => DAT_40021018
6c1a  42 f0  orr   r2, r2, #0x20
      20 02
6c1e  9a 61  str   r2,[r3,#0x18] => DAT_40021018
6c20  9b 69  ldr   r3,[r3,#0x18] => DAT_40021018
```

LISTING 4: MMIO addresses found in function containing multiple LMS patterns in ARM Cortex-M assembly, where load address outside of the LMS pattern are also MMIO addresses.

to reference MMIO addresses. This is logical because more LMS patterns in a function suggest that the function's primary purpose is interacting with peripherals. So the auxiliary store and load instructions are also highly likely to communicate with the peripherals.

Proximity: The third pattern we identified is referred to as Proximity. When firmware exchanges data with peripherals, it loads the peripheral's data into a register for later use. Because MMIO addresses are volatile [53, 54], data that is loaded in succession results in different data. Another reason why peripherals are accessed in succession is when multiple configurations need to be set. A simple example of this is when we want to enable a yellow colour on an RGB light. To do this, we first need to enable a bit so the light becomes green, and then a second bit to make it yellow. For both the load and store cases, we observe that the assembly accesses the same load or store instructions sequentially, as shown in listing 5. What we noticed is that when we group many of the load (or store) instructions, we are more certain that the address they reference is an MMIO address. This makes sense because a large number of load instructions are sometimes needed when exchanging data, while some peripherals have multiple bits in a configuration that need to be set to work correctly.

```
199a 9a  6a   ldr    r2,[r3,#0x28 ]=> DAT_e000ed28
199c 9a  6a   ldr    r2,[r3,#0x28 ]=> DAT_e000ed28
199e 9a  6a   ldr    r2,[r3,#0x28 ]=> DAT_e000ed28
19a6 9a  6a   ldr    r2,[r3,#0x28 ]=> DAT_e000ed28
```

LISTING 5: The same MMIO address is referenced sequently in ARM Cortex-M assembly.

Store Instructions Variation (SIV): The fourth pattern we identified is referred to as SIV. When firmware boots up, multiple interactions are initiated with peripherals. One of these interactions involves setting the configuration of the peripherals. Since the CPU is connected to multiple peripherals, various configurations need to be set sequentially before the main operation executes. We observe in the assembly that, for efficiency, this is achieved by storing the same data (where possible) to different peripherals. Specifically, the register to which the data is sent differs, while the register holding the data remains the same. This pattern can be seen in listing 6. If we group many of these instructions, we become more certain that the addresses they reference are MMIO addresses. Our certainty increases because multiple peripherals exist on an MCU, implying that there are also multiple configuration addresses. As some configuration addresses need to be set as soon as possible before the main execution begins, it is logical that a group consisting of many of these instructions is about the configuration.

```
2e90 33  60   str    r3,[r6,#0x0 ]=> DAT_40038014
2e92 13  60   str    r3,[r2,#0x0 ]=> DAT_4003801c
2e94 2b  60   str    r3,[r5,#0x0 ]=> DAT_40038024
2e96 23  60   str    r3,[r4,#0x0 ]=> DAT_4003802c
2e98 3b  60   str    r3,[r7,#0x0 ]=> DAT_40038034
```

LISTING 6: Multiple MMIO addresses that are in proximity where the register that data is send to is different, while the other register and the offset stay the same in ARM Cortex-M assembly.

Offset: The final pattern we identified is referred to as Offset. This pattern differs from the patterns mentioned before, as it utilises previously located MMIO addresses to locate new ones. What we observed is that ARM Cortex-M and Xtensa firmware use a register in combination with an offset for their store and load operations. We can utilise this characteristic to locate new MMIO addresses using previously located MMIO addresses. This can be done by removing the offset from a previously located MMIO address. After which, we search through the assembly for addresses that use the same base address. If a matching base address is identified, we add the offset of the respective instruction to the base address to locate the new MMIO address. The mentioned offset logic can be seen in listing 7.

```
1cc2 1a  62   str    r2,[r3,#0x20 ]=> DAT_40023820
1cd0 5a  62   str    r2,[r3,#0x24 ]=> DAT_40023824
1cde 1a  61   str    r2,[r3,#0x10 ]=> DAT_40023810
1cec 5a  61   str    r2,[r3,#0x14 ]=> DAT_40023814
1cfa 9a  61   str    r2,[r3,#0x18 ]=> DAT_40023818
```

LISTING 7: MMIO addresses where the base address is the same but the offset is different in ARM Cortex-M assembly.

To determine the best combination of the previously mentioned patterns, we perform a ground truth analysis in section 5.3. For this analysis, we tested all possible combinations of the identified patterns. Among these, three patterns require numerical parameters to perform their functionality. We chose the parameter ranges defined below, as higher values did not alter our results, while lower values defeated the purpose of the patterns.

- **FRI** has an integer range from 1 to 6. An integer of 1 means that only one LMS is present in a function, while, for example, 6 means there are six LMS patterns present in a function.
- **Proximity and SIV** have two parameters. In both cases, the first parameter defines how many proximity instructions are near each other. This range goes from 2 to 6. An integer of 2 indicates that two load (or store) instructions are close to each other, while 6 indicates that there are six. The second parameter relates to what near means. It defines the bytes between the assembly instructions. For this, we select a range from 0 to 64, where 0 means that the instructions are immediately adjacent, while 64 means there are 64 bytes separating them.

4 Pipeline design

The above patterns are integrated into a pipeline, which is illustrated in fig. 3. This pipeline consists of three components, which are explained below.

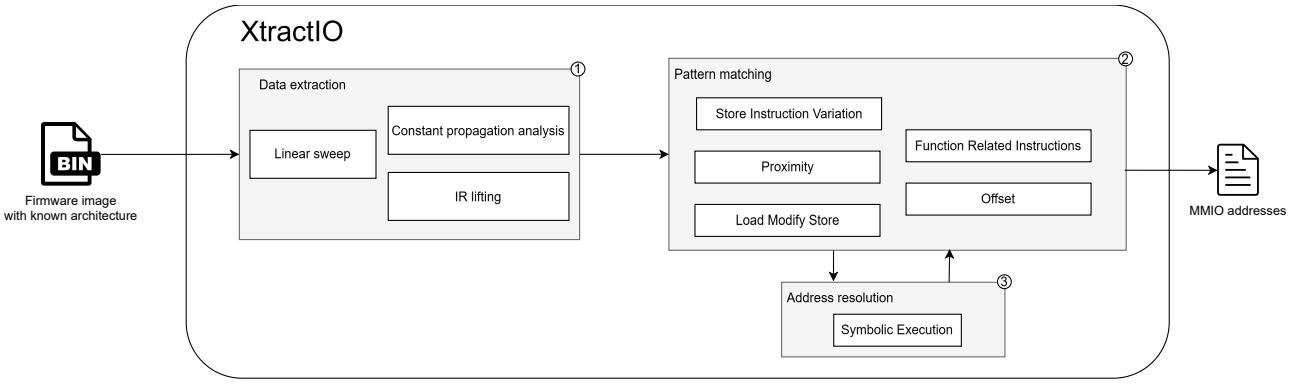


FIGURE 3: XtractIO overview. Our static analysis automatically locates MMIO addresses for ARM Cortex-M and Xtensa firmware by processing instructions using identified patterns.

① **Data Extraction:** After the pipeline receives a firmware image with its architecture, it first performs data extraction. This data extraction involves first performing a linear sweep over the data to decode the bytes of the instructions [55]. These decoded bytes will then be used as input for our IR lifting and constant propagation analysis. We lift our decoded bytes to an IR because we want to generalise our analysis instead of making a specific algorithm for each architecture. Constant propagation analysis, which replaces variables with known constants, is used to extract the referenced memory addresses from the firmware where possible. The IR and referenced memory addresses are the output of this component.

② **Pattern matching:** When the pattern matching component receives the IR and referenced memory addresses, the pipeline starts its pattern matching. The IR matches with our identified patterns to locate MMIO addresses. Before matching with our Offset pattern, the pipeline sends the instruction first to the address resolution component ③ to resolve some missing referenced addresses. After this component receives the updated data, it matches with the final Offset pattern. This component outputs the MMIO addresses located by the pipeline.

③ **Address resolution:** As the final component of our pipeline, we have the address resolution. This component resolves addresses using symbolic execution that were not resolved by component ②, due to, indirect jumps, function returns, etc. These resolved addresses will then be sent back as output to the pattern-matching component.

4.1 Implementation

In this subsection, we will explain the implementation of the different components.

① **Data extraction:** We leverage two frameworks to run the linear sweep algorithm: Angr and Ghidra. We use Angr because, in our observation, it decodes more bytes than Ghidra, resulting in more disassembled code to analyse. Ghidra was selected because it supports a broader range of architectures. Following the linear sweep, we used Pypcode as the library for lifting the decoded bytes to IR. Pypcode was chosen for this because it supports the same architectures as the frameworks do. Besides the IR lifting, we also provided the decoded bytes as input to the constant propagation analysis. This analysis is only performed with Ghidra because, at the time of writing, Angr does not have an option for this.

② **Pattern matching:** For matching with our identified patterns, we use the IR instructions. An IR instruction can consist of multiple lines, as seen in, for example, listing 2. Each line has its own separate instruction called an opcode [56]. In the listing we have at line 1 the instruction (INTEGER_ADDITION) and at line 2 the instruction LOAD. Opcodes are used in the pipeline to locate MMIO addresses. We can identify with opcode if, for example, multiple load or store instructions are in proximity to each other, which points to the Proximity and SIV pattern. We can also identify with opcode if a load instruction is followed by a bitwise and store instruction, signifying an LMS pattern. When we have identified these patterns, we can also locate MMIO addresses through the Offset and FRI patterns. For the Offset pattern, we use the IR instructions as well as the referenced

Nr.	Pattern	FRI Threshold	SIV Threshold	Proximity threshold	Offset	Symbolic execution	unique
1	SIV, Proximity	-	34/2	50/3	No	No	No
2	SIV, Offset	-	34/2	-	Yes	No	No
3	LMS, FRI, SIV	3	56/2	-	No	No	No
4	LMS, FRI, SIV	6	54/4	-	No	No	No
5	SIV, Proximity, Offset	-	34/2	34/2	Yes	No	No
6	SIV, Proximity, Offset	-	34/6	8/5	Yes	No	No
7	LMS, FRI, SIV, Proximity	1	36/3	2/2	No	No	No
8	LMS, FRI, SIV, Proximity	6	40/6	4/6	No	No	No
9	LMS,FRI, SIV, Proximity, Offset	1	62/2	56/2	Yes	No	No
10	LMS,FRI, SIV, Proximity, Offset	1	62/2	54/2	Yes	No	Yes
11	LMS,FRI, SIV, Proximity, Offset	1	48/2	52/2	Yes	Yes	No
12	LMS,FRI, SIV, Proximity, Offset	1	8/4	2/2	LMS	Yes	No
13	LMS,FRI, SIV, Proximity, Offset	2	34/2	30/2	Yes	No	No
14	LMS,FRI, SIV, Proximity, Offset	3	34/6	22/4	Yes	No	No
15	LMS,FRI, SIV, Proximity, Offset	5	14/4	22/3	Yes	Yes	No
16	LMS,FRI, SIV, Proximity, Offset	5	34/6	10/5	Yes	No	No

TABLE 1: Table describing the different patterns with the used threshold and other auxiliary information. LMS in the offset column means that the Offset pattern is only used on the LMS pattern, in all other cases, it is used on all the patterns. Additionally, the unique column provides information on whether an address is counted once or multiple times, and whether it appears in the firmware more than once. Finally, in the SIV and Proximity columns, you see two parameters separated by a slash. The first one means the bytes between the instructions, while the second one means the number of instructions.

addresses as input. When we identify a pattern, we mark the referenced addresses that belong to the underlying instructions as MMIO. When an instruction does not have a referenced address, we send it to ③.

③ Address resolution: For resolving unknown referenced addresses of instructions that belong to a pattern, we use the symbolic execution engine of Angr with a time constraint. Symbolic execution is performed locally, starting at the function's entry point and continuing up to the instruction whose value needs to be resolved. As symbolic execution can take a long time to analyse certain cases, we limited it to 5 seconds per value resolution. This duration was chosen because MMIO addresses are generally constant, so they should be relatively quick to resolve, and as a practical trade-off between coverage and execution time. During analysis, we observed that 5 seconds was sufficient to resolve the majority of the referenced addresses. Increasing the timeout would significantly increase the duration of the pipeline's runtime, while providing only slightly more coverage. We did not automatically fine-tune this parameter, as this would not have been feasible because of time constraints. When the 5 second duration has been reached, the pipeline terminates the symbolic execution process for that resolution.

5 Evaluation

In this section, we first introduce our setup, including the hardware used, targets and MMIO segments. After this, we measure the performance of the pipeline.

5.1 Hardware

The hardware used for almost the entire project was a laptop containing an Intel Core i7-12700H (14 cores) processor with 48 GB of DDR4 RAM, a 2.70 GHz clock speed, and an SSD, running Windows as the OS.

For fine-tuning the SIV and Proximity parameters regarding the bytes between instructions, we used a different machine. This was the EEMCS High-Performance Computing (HPC) cluster, located at the University of Twente, containing an AMD EPYC 7302P (32 cores), 256GB RAM, 3 GHz clock speed, and a HDD raidz2.

5.2 Dataset

Our dataset used a collection containing two different architectures, ARM Cortex-M and Xtensa. We focus on these architectures because previous research shows that they make up the majority of non-Linux firmware [24]. For this collection, we sampled, for ARM Cortex-M, twenty images from the Monolithic Firmware Collection [57]. This sampling was done pseudorandomly to ensure a diverse set for our analysis. This means we did not choose two images that were similar in name, such as blink_led and blink_led_2. Another thing that we kept in mind while sampling was that the MCU of the sample can be identified. As we want to use this to find the corresponding

Category	filter	pattern	precision	recall	f1-score
Address specific	Best precision	pattern 8	89.26%	44.28%	55.61%
	Best recall	pattern 9	44.28%	63.0%	46.85%
	Best f1-score	pattern 7	71.7%	57.84%	61.72%
Segment specific	Best precision	pattern 6	63.27%	84.17%	66.92%
	Best recall	pattern 13	48.92%	91.25%	60.37%
	Best f1-score	pattern 6	63.27%	84.17%	66.92%

(A) ARM Cortex-M & Xtensa.

Category	filter	pattern	precision	recall	f1-score
Address specific	Best precision	pattern 8	94.78%	27.82%	41.44%
	Best recall	pattern 11	45.95%	60.26%	48.07%
	Best f1-score	pattern 12	70.52%	56.33%	61.76%
Segment specific	Best precision	pattern 16	74.58%	74.17%	70.48%
	Best recall	pattern 15	42.18%	93.33%	57.26%
	Best f1-score	pattern 14	71.25%	78.33%	71.5%

(B) ARM Cortex-M.

Category	filter	pattern	precision	recall	f1-score
Address specific	Best precision	pattern 4	83.78%	61.4%	70.24%
	Best recall	pattern 10	40.38%	77.11%	51.92%
	Best f1-score	pattern 3	78.94%	69.74%	73.6%
Segment specific	Best precision	pattern 1	58.55%	75.0%	60.79%
	Best recall	pattern 5	39.87%	100.0%	55.15%
	Best f1-score	pattern 2	52.37%	97.5%	64.4%

(C) Xtensa.

TABLE 2: Overview of the best-performing patterns for three architectures. Each table presents the pattern resulting in the highest precision, recall and F1-score for the address and segment-specific categories. Precision refers to the percentage of addresses we locate that are MMIO compared to non-MMIO. Recall refers to the percentage of MMIO addresses we locate compared to those we do not locate. The F1-score balances both. Patterns listed in the table refer to those in table 1.

datasheet. For Xtensa, we compiled twenty samples ourselves. For this, we used the Zephyr [58] and ESP IDF [59] frameworks. We did the compilation for three MCUs: ESP32, ESP32-S2, and ESP32-S3. Among these, we divided the twenty images evenly. All the samples we acquired can be found in appendix B. After acquiring our samples, we used the corresponding MCUs to locate their datasheets. From this datasheet, we extracted the segments that correspond to the MMIO segments. After we extracted this information, we performed a post-processing step. In this post-processing step, we extracted all the referenced addresses from the firmware and cross-referenced them with the datasheet segments to identify which segments are used by our samples. We did this because we noticed that not all segments described on the memory map are used. An example of this is that the sample st-plc has on its MCU datasheet a segment `0x5000000-0x5fffffff` as part of its peripheral segments [60]. However, after our analysis, we identified that this sample does not use this segment. After we completed our postprocessing, we ended with the segments we use in our analysis below. Our defined MMIO segments can be found in appendix B.

5.3 Performance

We measure the accuracy of locating MMIO addresses against our dataset. For this, we use three performance metrics: recall, precision and F1-score. For the average F1-score, we used the macro F1-score, as we want to treat each sample equally [61].

The results are separated into two categories: address-specific and segment-specific. In the address-specific category, we focus on the individual addresses. This means that a false positive occurs when we locate a non-MMIO address, while a false negative occurs when we do not locate an MMIO address. Consequently, even a single address can affect the final result. The segment-specific category is broader, as it focuses on whether we successfully locate all segments that contain MMIO addresses, for example, `0x40000000-0x50000000` as a segment for ARM Cortex-M. This means that a false positive occurs when we locate one address within a non-MMIO segment, while a false negative occurs when we miss all the addresses of an MMIO segment. The two categories can be found in table 2. This table also shows us the patterns that result in the least amount of false positives (precision), false nega-

Category	architecture	precision	recall	f1-score
Address specific	ARM Cortex-M & Xtensa	26.22%	100%	38.97%
	ARM Cortex-M	32.18%	100%	45.90%
	Xtensa	20.26%	100%	32.03%
Segment specific	ARM Cortex-M & Xtensa	17.20%	100%	28.59%
	ARM Cortex-M	21.75%	100%	35.12%
	Xtensa	12.65%	100%	22.06%

TABLE 3: Overview of the naive approach performance. The table presents the precision, recall and F1-score for the different architectures across the address and segment-specific categories.

Category	architecture	median f1-score
Address specific	ARM Cortex-M & Xtensa	61.86%
	ARM Cortex-M	66.24%
	Xtensa	79.99%
Segment specific	ARM Cortex-M & Xtensa	66.67%
	ARM Cortex-M	73.34%
	Xtensa	66.67%

TABLE 4: The median macro f1-score for the different architectures separated over address and segment-specific categories.

tives (recall), and the resulting F1-score, which balances both.

5.3.1 Naive approach

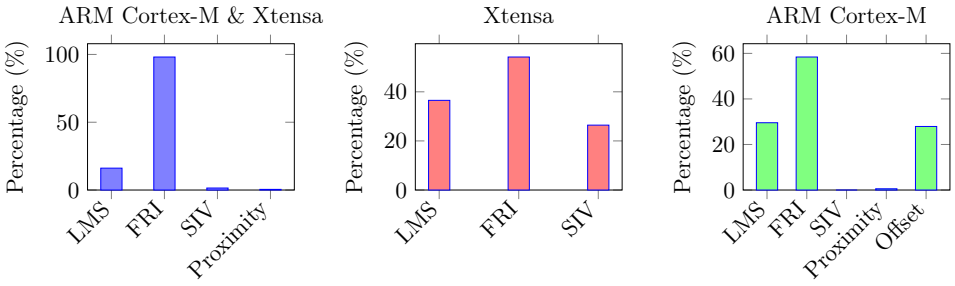
We compare our approach against a benchmark naive approach. For the naive approach, we take all the located referenced addresses as input and calculate our performance without any filtering, as shown in table 3. Compared to the naive approach, XtractIO is on average 30 to 40 per cent better. It was to be expected that our approach would be higher than a naive approach, as taking all addresses as input tends to get numerous false positives.

5.3.2 Different categories

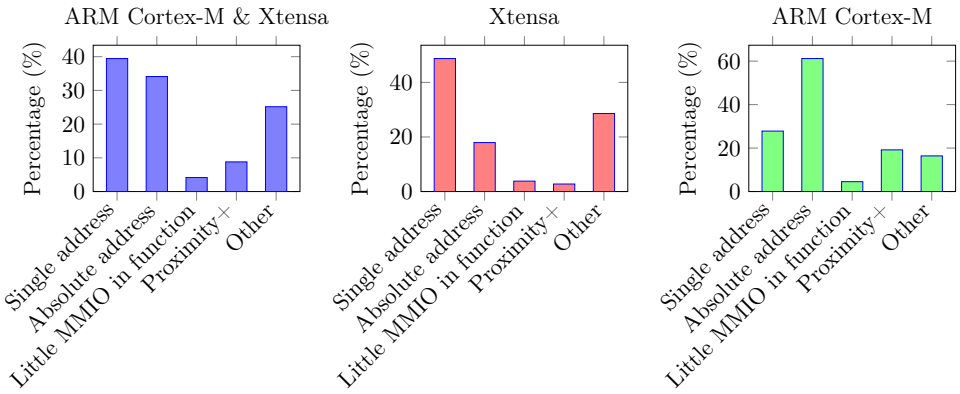
The result of the average best F1-score, which optimally balances the trade-off between false positives and false negatives, shows us in table 2 that we get fewer false negatives than false positives for the segment-specific category. In contrast, for the address-specific category, the situation is reversed. These results are as expected because for the segment-specific category, we only have a limited number of segments that need to be matched. This means that if we match a segment, then our false negative rate decreases drastically. However, because of these limited segments, it is also easier to increase the false positive rate more quickly. This is because if one address matches a non-MMIO segment, it already counts as a false-positive for that entire segment. Therefore, if we have, for example, sixteen segments where one is MMIO and the rest are non-MMIO, then matching with one non-MMIO segment can already give us 50 per cent false positives. In contrast, when we examine the address-specific category, we do not have this small group of segments. Here we have a far larger group consisting of all the individual referenced addresses. Therefore, it is easier to miss multiple addresses for the address-specific category than for the segment-specific category, resulting in a higher number of false negatives. However, having this large group also means that we can have a lot of MMIO addresses compared to non-MMIO addresses. This results in a lower false positive rate than for the segment-specific category.

5.3.3 Median F1-score

Comparing our average best F1-score in table 2 with the median displayed in table 4, we observe that generally for both the address and segment-specific categories, the median is higher than its respective average, ranging from a few per cent behind the comma to around six per cent. Breaking down our average results into individual samples, as shown in appendix C, we get an idea of why this is the case. The individual results show us that our patterns are more effective in some cases and less effective in others. Notably, our best address-specific results have an F1-score of around 90 per cent, while our lowest scores average around 27 per cent. This difference is also evident in the segment-specific results, as the highest scores are 100 per cent, while the lowest score averages around 22 per cent. Because our results contain a slightly larger number of low scores than high scores, we observe that our average is being lowered, resulting in an outcome where our median is higher than our average. The difference between our samples was not



(A) Average false positives for each architecture per pattern.



(B) Average false negatives for each architecture per category. The other category shows false positives that could not be linked to any of the other categories.

FIGURE 4: False positive and false negative distributions of the best F1-score for ARM Cortex-M, Xtensa and the combined address-specific categories. The percentages of the different patterns/categories are generally more than 100 per cent. This is due to some false positives (or negatives) contributing to more than one pattern/category.

what we expected because we cross-referenced our identified patterns across multiple samples. We expected that by using this technique, we would get results that are clustered closely together and not as spread out as they are now.

5.3.4 Average F1-score

When we examine the best F1-score, which provides the optimal trade-off between false positives and negatives on average for our results in table 2, we observe that they are around 60 to 70 per cent, indicating that we introduce both false positives and negatives. These scores are lower than we expected. After performing additional research, we identified multiple reasons why this is the case. These are described in the false positive and negative sections below, where the results of them are illustrated in fig. 4. In these subsections, we focus solely on the macro F1-score of the address-specific category. This is because we are interested in precisely what causes the false negatives and positives on the individual address level.

5.3.5 False positive

Our pipeline identifies false positives due to one reason. This is because our patterns are not solely used by MMIO addresses but also by non-MMIO addresses. If we split the false positives into the individual patterns that create them, we see a few things.

- **LMS:** This pattern locates false positives because it is a pattern used to clear and set bits directly. This kind of behaviour is not reserved only for peripherals, as we also see it in RAM. An example of this is for updating error codes, where to update the error code to a specific error, we see that this operation is also performed, as shown in listing 14.
- **FRI:** As mentioned in section 3, the FRI

pattern relies on the LMS pattern to identify functions that contain MMIO instructions. If LMS incorrectly identifies an MMIO as non-MMIO, it results in flawed input for the FRI. As a result, the FRI will capture all the load and store instructions from the function where this flawed LMS occurred, even though the instructions will reference non-MMIO addresses.

- **Proximity:** This pattern locates minimal false positives. However, the false positives it creates stem from the fact that multiple function calls are performed sequentially, where before the function call, the same register is loaded with the same data. This is done, for example, when the same function argument is used for multiple function calls.

- **SIV:** This pattern locates false positives because the bytes between instructions get too big, for a group that is too small. We observe that the SIV's false positives are primarily found in Xtensa. When we look at the result in table 2 with the corresponding patterns described in table 1. We observe that Xtensa achieves its best F1-score, where the trade-off of false positives and negatives is minimised, when the bytes between instructions are set to 56 and the group size is set to just two. During our analysis, we observed that these parameters are the problem, as it is not uncommon for the same register to store data to two different registers within a sufficiently wide byte space.

- **Offset:** Finally, we see that the Offset pattern also locates multiple false positives on average for only ARM Cortex-M. It is expected that this pattern locates false positives. This is because RAM also uses the pattern where multiple addresses share the same base address. This means that when the Offset pattern receives a non-MMIO address as input, it identifies its base address and searches through the code for addresses that use the same base address, which are also non-MMIO addresses.

Although the patterns locate false positives, we also observe that they locate a significant number of true positives. To be precise 60 to 70 per cent of the memory addresses located are true positives. This indicates that the patterns we used are to a certain extent effective. To ensure that the patterns work even better, we must define parameters logically so that arbitrary read/write operations cannot be incorporated. Additionally, creating anti-patterns which locate non-MMIO addresses also helps, as this can be used to suppress the false positives for the patterns.

5.3.6 False negative

In contrast to what we described for the false positives, we have multiple reasons why MMIO addresses are not found. We have separated these into different categories and are listed below.

- **Single addresses:** We identified that when an address is only referenced once in the entire firmware, we do not detect it. This is because four of our patterns locate an MMIO address using multiple addresses as input, if an address is only referenced once, we do not have any auxiliary information to correlate it with. Additionally, when these single addresses do not share a base address with the addresses we do locate, we also cannot use our Offset pattern to locate them. This is because this pattern examines shared base addresses to identify new addresses. The results show that, on average, this gives approximately 49 per cent of the false negatives identified for Xtensa. While for ARM Cortex-M, this gives us 28 per cent.

- **Absolute addresses:** We also identified that multiple addresses are referenced with only an absolute load and not a register-relative load, an example of the difference is seen in listing 8. This category results in 60 per cent false negatives for ARM Cortex-M, while around 18 per cent for Xtensa. That an address is only referenced through an absolute load happens when the address itself is used in e.g. a comparison or for calculating dynamic addresses. Our patterns are only focused on the register-relative load. This results in us not being able to locate these addresses.

```
1    129e6 27 4b  ldr   r3,[DAT_00012a84]       = 40023800h
2    129e8 5b 6c  ldr   r3,[r3,#0x44]=>DAT_40023844
```

LISTING 8: The first instruction illustrates an absolute load while the second instruction illustrates a register relative load for ARM Cortex-M.

- **Little MMIO in a function:** In some cases, we observe that MMIO addresses are located in a function that consists primarily of non-MMIO addresses. We struggle to detect these, as the patterns locate MMIO addresses using multiple addresses as input. When an MMIO address is in a function that has few to no MMIO addresses, we cannot use these patterns.
- **Proximity+:** Another thing we observe is that firmware loads or stores data to different MMIO addresses that share the same base address in quick succession. This logic can, for example, happen when the same value needs to be stored to multiple configuration addresses or when configuration data from multiple registers needs to be loaded. We do not detect this because we only focus on the recurrence of one address. However, we do note that this logic is also often used by non-MMIO addresses.

6 Discussion

In this section, we will discuss our previous findings.

Our results show that we can reliably locate MMIO addresses and MMIO segments for some samples. This is, for example, the case for the sample button_s3 when looking at the address specific category and for the sample p2im_reflow_oven for the segment specific category. Using our pipeline, we get almost 85 per cent F1-score for button_s3. Showing that the results we got are mostly MMIO addresses with only a few MMIO addresses missing. Using our tool on p2im_reflow_oven, we see that we get an F1-score of 100 per cent, meaning that it successfully found all segments containing MMIO addresses without misidentifying non-MMIO segments.

The result shows us that we can use our pipeline to create an initial scope for finding which segment contains MMIO addresses. This should then be fine-tuned by another tool to remove the false positives that are present. We can also use our pipeline as input for manual analysis. Although it does not locate all addresses, we eliminate a significant number of false positives. This means that if someone performs manual analysis on our results for further processing, they can obtain a subset of MMIO addresses more quickly compared to manually analysing the entire firmware.

Unfortunately, even though our pipeline can perform well for the approaches above, we will be less likely to use it for an approach where accuracy is crucial, e.g. fuzzing. For fuzzing, we want to focus only on the MMIO addresses. Our results show that our pipeline does not locate all MMIO addresses. Because of this, the fuzzer will miss vulnerabilities when using our data as input. Additionally, our pipeline also locates non-MMIO addresses, which are less interesting to fuzz compared to MMIO addresses. Another approach our pipeline is less likely to be used for is re-hosting. If we want, for example, to build a memory map of the peripherals using our output as input, which is then used for an emulator, we get undefined behaviour.

When we examine the different patterns used for our results, as shown in table 1, we see that all identified patterns help locate MMIO addresses. Looking at it in more depth, we observe that for eight of our results, we use all the identified patterns. If we look at it more broadly, we see that we use the LMS, FRI and SIV patterns in thirteen of them. We observe that when all patterns are used, we generally obtain the lowest number of false negatives. This applies to both Xtensa (only address specific) and ARM Cortex-M, as well as the combined result. However, these patterns also show a high false positive rate, indicating that these approaches also locate multiple non-MMIO addresses. To improve this, we can identify anti-patterns, which are, in this case, patterns that locate typical non-MMIO addresses. With this approach, we will get our false positive rate to a desired level.

Another thing that our results in table 2 show is that our patterns can locate the MMIO segment for ARM Cortex-M better than for Xtensa. When we want to locate the MMIO addresses themselves, then our patterns are doing better for Xtensa. The

reason why the performance for locating individual MMIO addresses in ARM Cortex-M is lower than that of Xtensa is that the results we obtain contain more non-MMIO addresses than those for Xtensa, leading to a higher false positive rate. Additionally, our patterns locate fewer MMIO addresses for ARM Cortex-M than for Xtensa, resulting in more false negatives. Combining these two means that the combined result is also lower. The reason why the performance for locating segments in Xtensa is lower than for ARM Cortex-M is that the segments Xtensa has containing MMIO addresses are smaller than those of ARM Cortex-M. This means that more segments exist than for ARM Cortex-M. As a direct result, it is easier for performance to decline because addresses can be found in more segments, while it is also easier to miss segments.

Another interesting observation is that when we combine ARM Cortex-M and Xtensa as shown in table 2, we see that the combined results do not precisely fall between the individual results. When we look at this some more, we notice that different patterns are generally used for this combined result. This suggests that one pattern is more optimal for locating MMIO addresses when we look at multiple architectures simultaneously, while another pattern is more optimal when looking at architectures one at a time.

Limitations: The first limitation of our pipeline is that it only supports the architectures that are provided by the frameworks [62]. Even though there are many architectures in here, if there is one that is not provided by it, then we cannot use our pipeline to locate MMIO addresses.

The second limitation we face is related to our symbolic execution method. As said in section 4.1, we stop our symbolic execution after 5 seconds, as referenced MMIO addresses are generally constant. However, it is possible that the symbolic execution gets temporarily stuck in a loop or encounters a state explosion. This can result in that 5 seconds is not enough to resolve the addresses, even though they are constant.

Our third limitation is that we examined patterns in ARM Cortex-M firmware and tried to match them to Xtensa firmware, which in this work was not a problem, as we focused on ARM Cortex-M and Xtensa. However, if we want to extend it to, for example, RISC-V or proprietary firmware, we need to ensure that we cross-reference the found patterns with more architectures to verify they also work on these.

Our final limitation is that we rely on the Ghidra headless mode for some of our results. However, during some experimentation, we noticed minor dissimilarities between running our code through headless mode and through the Ghidra script manager from the GUI. The dissimilarities we encountered were that some functions were not found through the headless mode, while the GUI did locate them. As we were unable to find a way to run our code outside of headless mode, we note this as a limitation.

Future work: In the future, our work can be expanded upon in three directions: enrichment, generalisation and higher-level analysis.

As shown before, our approach works better than a naive approach. However, we still need to enrich our pipeline with more patterns to ensure that our false negatives decrease, while we need to identify anti-patterns to reduce our false positives.

Additionally, we focused in this paper on ARM Cortex-M and Xtensa. These two are architectures that are prominently seen in non-Linux firmware [24]. However, they are not the only ones. Taking more architectures into account, like RISC-V and proprietary ones, will be another direction we can work towards in the future.

Finally, the patterns that we identified are focused on what happens inside a function. However, functions also interact with each other. These interactions can also provide valuable insights into the working of MMIO addresses, which can help locate them.

7 Related Work

MMIO addresses can be located in several ways. In this section, we have grouped the works that share a commonality in their methods for locating MMIO addresses. In general, current research focuses on the ARM Cortex-M architecture for locating MMIO addresses.

7.1 MCU documentation

A common approach is to manually look up MMIO segments from the datasheet [7–12, 17–21]. Besides looking them up, Gustafson et al. also has special rules to classify how data communicates with peripherals, allowing them to track IO operations [17]. Zhou et al. use the datasheet to

manually extract peripheral register tables using Python libraries such as Tabula [22]. Using this, they create textual descriptions of the MMIO registers that can be used as input for their approach.

Compared to these approaches, our approach does not rely on an external datasheet to look up or extract MMIO addresses from. We only use the information coming from the firmware itself. Additionally, Gustafson et al. have a rule called "read-modify-write". This is a similar rule to our LMS pattern, with as difference that our pattern is more specific by limiting the modify instruction between the load and store to one bitwise operation.

Similarly to the datasheet, manufacturers share files such as System View Description and Host-Specific Development Tool Libraries, which also contain information regarding the MMIO addresses. Kim et al. programmatically parse through these files to look up the different MMIO segments a peripheral has [23].

Like before, this approach also relies on documentation to work. While for our approach, this is not necessary.

7.2 Static analysis

Some works use static analysis instead of relying on the MCU documentation. Wen et al., for example, use the knowledge that a Software Development Kit (SDK) function will eventually read or write to MMIO addresses [28]. Using this knowledge, they monitor the data flow and identify the locations of the MMIO addresses.

While Wen et al. focus on specific SDK functions, our approach does not focus on a particular function and applies patterns to all the functions present in the firmware.

Shen et al. focus solely on constant pointers. They locate MMIO addresses by assuming that all memory instructions with constant pointer operands are MMIO accesses [63].

This work differs from ours in that we use patterns to locate MMIO addresses, whereas this approach examines constant pointer operands to locate MMIO addresses.

7.3 Dynamic analysis

Gui et al. use a firmware emulator which includes hooks that monitor read and write operations to unmapped memory addresses during emulation [42]. If an exception is triggered outside of their predefined memory segments, then their system records this as peripheral access and captures the peripheral type and memory addresses accessed. Unfortunately, it is not clear from this work how the predefined memory segments are extracted.

Instead of doing dynamic analysis. Our work focuses solely on static analysis, removing the need for an emulator. Additionally, we do not need to know the predefined memory segments beforehand.

8 Conclusion

In this paper, we introduced XtractIO, a way to statically locate MMIO addresses from ARM Cortex-M and Xtensa firmware. This is the first research done for automatic MMIO identification across multiple architectures, to the best of our knowledge. We have shown that MMIO addresses have multiple patterns which can be used to locate them. Another thing we have shown is that by applying our pipeline, we can better locate MMIO addresses than by using a naive approach. However, still improvements can be made by, for example, identifying patterns for MMIO addresses that only occur once in a firmware, taking into account addresses that only have absolute references and locating MMIO addresses that appear in functions that contain mostly non-MMIO addresses. In the end, this paper shows that automatically locating MMIO addresses is possible, but improvements still need to be made to make it reliable.

9 Acknowledgements

I want to thank my supervisors, Jorik van Nielen and Andrea Continella, for their guidance throughout the thesis period.

During my thesis, ChatGPT and Grammarly assisted me with my work. ChatGPT was used to help debug some code, help write functions, and repair a handful of broken sentences, while Grammarly was used as a grammar checker. All the changes resulting from these two tools were double-checked by me.

References

1. *Embedded Systems Market Size, Share & Growth | Report [2030]* Oct. 2024. https://www.fortunebusinessinsights.com/embedded-systems-market-108767.
2. *The Past, Present and Future of Cybersecurity for Embedded Systems* 2020. https://blackberry.qnx.com/content/dam/bbcomv4/qnx/resource-center/pdf/Cybersecurity%20for%20Embedded%20Systems%20-%20QNX%20Whitepaper.pdf.
3. Serper, Y. *'FriendlyName' Buffer Overflow Vulnerability in Wemo Smart Plug V2* May 2023. https://sternumiot.com/iot-blog/mini-smart-plug-v2-vulnerability-buffer-overflow/.
4. Zieniūtė. *What is the Mirai botnet, and how does it spread?* Apr. 2024. https://nordvpn.com/nl/blog/mirai-botnet/.
5. Naraine. *GreyNoise Credits AI for Spotting Exploit Attempts on IoT Livestream Cams* Nov. 2024. https://www.securityweek.com/greynoise-credits-ai-for-spotting-exploit-attempts-on-iot-livestream-cams/.
6. *Fuzz Testing* 2020. https://www.blackduck.com/glossary/what-is-fuzz-testing.html.
7. Scharnowski, T. *et al.* Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing. en.
8. Kim, J., Yu, J., Lee, Y., Kim, D. D. & Yun, J. HD-FUZZ: Hardware Dependency-Aware Firmware Fuzzing via Hybrid MMIO Modeling. en.
9. Gustafson, E. *et al.* Toward the Analysis of Embedded Firmware through Automated Re-hosting. en.
10. Farrelly, G., Chesser, M. & Ranasinghe, D. C. *Ember-IO: Effective Firmware Fuzzing with Model-Free Memory Mapped IO* en. arXiv:2301.06689 [cs]. Jan. 2023. http://arxiv.org/abs/2301.06689 (2024).
11. Scharnowski, T., Wörner, S., Buchmann, F. & Holz, T. Hoedur: Embedded Firmware Fuzzing using Multi-Stream Inputs. en.
12. Johnson, E., Mason, J., College, O. & Savage, S. Jetset: Targeted Firmware Rehosting for Embedded Systems. en.
13. Murray, J. M. & Wiatrowski, C. A. Microcomputer Peripherals. *IEEE Transactions on Industrial Electronics and Control Instrumentation* **IECI-25,** 303–322 (1978).
14. Sarangi, S. R. *Basic Computer Architecture* 1st edition. ISBN: 1636403034 (White Falcon Publishing, Sept. 2021).
15. Summerville, D. Embedded Systems Interfacing for Engineers using the Freescale HCS08 Microcontroller II: Digital and Analog Hardware Interfacing. *Synthesis Lectures on Digital Circuits and Systems* **4,** 1–139 (Aug. 2009).
16. Khan, A., Kim, H., Lee, B., Xu, D. & Bianchi, A. M2MON: Building an MMIO-based Security Reference Monitor for Unmanned Vehicles. en.
17. Gustafson, E. *et al. Shimware: Toward Practical Security Retrofitting for Monolithic Firmware Images* en. in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses* (ACM, Hong Kong China, Oct. 2023), 32–45. ISBN: 9798400707650. https://dl.acm.org/doi/10.1145/3607199.3607217 (2024).
18. Cao, C., Guan, L., Ming, J. & Liu, P. *Device-agnostic Firmware Execution is Possible: A Concolic Execution Approach for Peripheral Emulation* en. in *Annual Computer Security Applications Conference* (ACM, Austin USA, Dec. 2020), 746–759. ISBN: 978-1-4503-8858-0. https://dl.acm.org/doi/10.1145/3427228.3427280 (2024).
19. Chesser, M. MultiFuzz: A Multi-Stream Fuzzer For Testing Monolithic Firmware. en.
20. Feng, B., Mera, A. & Lu, L. P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. en.
21. Zhou, W. Automatic Firmware Emulation through Invalidity-guided Knowledge Inference. en.

22. Zhou, W., Zhang, L., Guan, L., Liu, P. & Zhang, Y. *What Your Firmware Tells You Is Not How You Should Emulate It: A Specification-Guided Approach for Firmware Emulation* en. in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (ACM, Los Angeles CA USA, Nov. 2022), 3269–3283. ISBN: 978-1-4503-9450-5. https://dl.acm.org/doi/10.1145/3548606.3559386 (2024).

23. Kim, T. *et al.* PASAN: Detecting Peripheral Access Concurrency Bugs within Bare-Metal Embedded Applications. en.

24. Nino, N. *et al. Unveiling IoT Security in Reality: A Firmware-Centric Journey* in *33rd USENIX Security Symposium (USENIX Security 24)* (USENIX Association, Philadelphia, PA, Aug. 2024), 5609–5626. ISBN: 978-1-939133-44-1. https://www.usenix.org/conference/usenixsecurity24/presentation/nino.

25. *What Is Firmware? Types and Examples* https://www.fortinet.com/resources/cyberglossary/what-is-firmware.

26. Muench, M., Stijohann, J., Kargl, F., Francillon, A. & Balzarotti, D. *What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices* en. in *Proceedings 2018 Network and Distributed System Security Symposium* (Internet Society, San Diego, CA, 2018). ISBN: 978-1-891562-49-5. https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_01A-4_Muench_paper.pdf (2024).

27. Eager, M. J. *Introduction to the DWARF Debugging Format* in (2007). https://api.semanticscholar.org/CorpusID:14261900.

28. Wen, H., Lin, Z. & Zhang, Y. *FirmXRay: Detecting Bluetooth Link Layer Vulnerabilities From Bare-Metal Firmware* en. in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (ACM, Virtual Event USA, Oct. 2020), 167–180. ISBN: 978-1-4503-7089-9. https://dl.acm.org/doi/10.1145/3372297.3423344 (2024).

29. Liu, J. *et al. RISC-Q: A Generator for Real-Time Quantum Control System-on-Chips Compatible with RISC-V* 2025. arXiv: 2505.14902 [cs.AR]. https://arxiv.org/abs/2505.14902.

30. Oshana, R. in *DSP Software Development Techniques for Embedded and Real-Time Systems* (ed Oshana, R.) 159–227 (Newnes, Burlington, 2006). ISBN: 978-0-7506-7759-2. https://www.sciencedirect.com/science/article/pii/B9780750677592500089.

31. Yu, T., Qu, X. & Cohen, M. B. *VDTest: an automated framework to support testing for virtual devices* Austin, Texas, 2016. https://doi.org/10.1145/2884781.2884866.

32. *Datasheet - stm32f103x8 stm32f103xb* Sept. 2023. https://www.st.com/resource/en/datasheet/stm32f103c8.pdf.

33. *Emulation* https://www.imaginationtech.com/glossary/emulation/.

34. *Interrupts* Sept. 2024. https://www.geeksforgeeks.org/interrupts/.

35. https://cs.lmu.edu/~ray/notes/ir/.

36. Click, C. & Paleczny, M. A simple graph-based intermediate representation. *SIGPLAN Not.* **30,** 35–49. ISSN: 0362-1340. https://doi.org/10.1145/202530.202534 (Mar. 1995).

37. Johnson. *Intermediate Representation* July 2010. https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/230%20Intermediate%20Rep.pdf.

38. Rouse, M. *Disassembler* Dec. 2011. https://www.techopedia.com/definition/6860/disassembler.

39. Cifuentes, C. & Gough, K. J. Decompilation of binary programs. *Software: Practice and Experience* **25,** 811–829. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380250706. https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380250706 (1995).

40. Aldrich, J. & Goues, C. L. Lecture Notes: Symbolic Execution. en.

41. Baldoni, R., Coppa, E., D'elia, D. C., Demetrescu, C. & Finocchi, I. A Survey of Symbolic Execution Techniques. en. *ACM Computing Surveys* **51,** 1–39. ISSN: 0360-0300, 1557-7341. https://dl.acm.org/doi/10.1145/3182657 (2024) (May 2019).

42. Gui, Z., Shu, H. & Yang, J. *FIRMNANO: Toward IoT Firmware Fuzzing Through Augmented Virtual Execution* in *2020 IEEE 11th International Conference on Software Engineering and Service Science (ICSESS)* ISSN: 2327-0594 (Oct. 2020), 290–294. https://ieeexplore.ieee.org/document/9237719/?arnumber=9237719 (2024).

43. Feng, X. *et al. Snipuzz: Black-box Fuzzing of IoT Firmware via Message Snippet Inference* en. in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (ACM, Virtual Event Republic of Korea, Nov. 2021), 337–350. ISBN: 978-1-4503-8454-4. https://dl.acm.org/doi/10.1145/3460120.3484543 (2024).

44. Seidel, L., Maier, D. & Muench, M. Forming Faster Firmware Fuzzers. en.

45. Malmain, R., Fioraldi, A. & Francillon, A. LibAFL QEMU: A Library for Fuzzing-oriented Emulation. en.

46. *What is fuzz testing?* https://about.gitlab.com/topics/devsecops/what-is-fuzz-testing/.

47. *Race Condition Vulnerability* Aug. 2024. https://www.geeksforgeeks.org/race-condition-vulnerability/.

48. Varma, R. *A Deep Dive into Static vs Dynamic Code Analysis* Dec. 2023. https://www.buildpiper.io/blogs/a-deep-dive-into-static-vs-dynamic-code-analysis/.

49. *Software engineering-control-flow-graph-cfg* Nov. 2024. https://www.geeksforgeeks.org/software-engineering-control-flow-graph-cfg/https://www.geeksforgeeks.org/software-engineering-control-flow-graph-cfg/.

50. *Data flow analysis in Compiler* Oct. 2024. https://www.geeksforgeeks.org/data-flow-analysis-compiler/.

51. *Data flow analysis: an informal introduction* https://clang.llvm.org/docs/DataFlowAnalysisIntro.html.

52. Zhang, P. in *Advanced Industrial Control Technology* (ed Zhang, P.) 613–683 (William Andrew Publishing, Oxford, 2010). ISBN: 978-1-4377-7807-6. https://www.sciencedirect.com/science/article/pii/B9781437778076100166.

53. *Understanding the "Memory Mapped I/O" and Volatile Qualifier* https://learningmicro.wordpress.com/understanding-the-memory-map-of-peripherals/.

54. Maldini, M. *Memory Mapped I/O in C* July 2024. https://www.embeddedrelated.com/showarticle/1683.php?.

55. Prasad, M. *Disassembly Challenges* Apr. 2003. https://www.usenix.org/legacy/publications/library/proceedings/usenix03/tech/full_papers/prasad/prasad_html/node5.html.

56. *Opcodes and operands* https://www.bbc.co.uk/bitesize/guides/z6x26yc/revision/4.

57. *monolithic-firmware-collection* https://github.com/ucsb-seclab/monolithic-firmware-collection.

58. *Zephyr Project* https://www.zephyrproject.org/.

59. *ESP-IDF Programming Guide* https://docs.espressif.com/projects/esp-idf/en/v4.2.2/esp32/index.html.

60. *STM32F427xx STM32F429xx* May 2025. https://www.st.com/resource/en/datasheet/stm32f427vg.pdf.

61. Leung, K. *Micro, Macro & Weighted Averages of F1 Score, Clearly Explained* Jan. 2023. https://www.kdnuggets.com/2023/01/micro-macro-weighted-averages-f1-score-clearly-explained.html.

62. *Architecture Support* https://api.angr.io/projects/pypcode/en/latest/languages.html.

63. Shen, M., Davis, J. C. & Machiry, A. *Towards Automated Identification of Layering Violations in Embedded Applications (WIP)* en. in *Proceedings of the 24th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems* (ACM, Orlando FL USA, June 2023), 143–147. ISBN: 9798400701740. https://dl.acm.org/doi/10.1145/3589610.3596271 (2024).

A Intermediate representation of patterns

```
IMARK ram[129e8:2]
unique[92600:4] = r3 + 0x44
r3 = *[ram]unique[92600:4]

IMARK ram[129ea:4]
unique[87700:4] = 0x10
shift_carry = CY
r3 = r3 | unique[87700:4]
tmpCY = shift_carry
tmpOV = OV
tmpNG = r3 s< 0x0
tmpZR = r3 == 0x0


IMARK ram[129ee:2]
unique[92600:4] = r2 + 0x44
*[ram]unique[92600:4] = r3
```

Listing 9: The load, modify, store pattern in pypcode IR.

```
IMARK ram[199a:2]
unique[98000:4] = r3 + 0x28
r2 = *[ram]unique[98000:4]

IMARK ram[199c:2]
unique[98000:4] = r3 + 0x28
r2 = *[ram]unique[98000:4]

IMARK ram[199e:2]
unique[98000:4] = r3 + 0x28
r2 = *[ram]unique[98000:4]

IMARK ram[19a6:2]
unique[98000:4] = r3 + 0x28
r2 = *[ram]unique[98000:4]
```

Listing 10: The same MMIO address is referenced sequently in pypcode IR.

```
IMARK ram[6bce:2]
unique[98000:4] = r3 + 0x18
r2 = *[ram]unique[98000:4]


IMARK ram[6bd0:4]
unique[87700:4] = 0x4
shift_carry = CY
r2 = r2 | unique[87700:4]
tmpCY = shift_carry
tmpOV = OV
tmpNG = r3 s< 0x0
tmpZR = r3 == 0x0


IMARK ram[6bd4:2]
unique[98000:4] = r3 + 0x18
*[ram]unique[98000:4] = r2
```

```
19
20 IMARK ram[6bd6:2]
21 unique[98000:4] = r3 + 0x18
22 r3 = *[ram]unique[98000:4]
23
24 IMARK ram[6be8:2]
25 unique[98000:4] = r3 + 0x18
26 r2 = *[ram]unique[98000:4]
27
28
29 IMARK ram[6bea:4]
30 unique[87700:4] = 0x8
31 shift_carry = CY
32 r2 = r2 | unique[87700:4]
33 tmpCY = shift_carry
34 tmpOV = OV
35 tmpNG = r3 s< 0x0
36 tmpZR = r3 == 0x0
37
38
39 IMARK ram[6bee:2]
40 unique[98000:4] = r3 + 0x18
41 *[ram]unique[98000:4] = r2
42
43 IMARK ram[6bf0:2]
44 unique[98000:4] = r3 + 0x18
45 r3 = *[ram]unique[98000:4]
46
47 IMARK ram[6c00:2]
48 unique[98000:4] = r3 + 0x18
49 r2 = *[ram]unique[98000:4]
50
51
52 IMARK ram[6c02:4]
53 unique[87700:4] = 0x10
54 shift_carry = CY
55 r2 = r2 | unique[87700:4]
56 tmpCY = shift_carry
57 tmpOV = OV
58 tmpNG = r3 s< 0x0
59 tmpZR = r3 == 0x0
60
61
62 IMARK ram[6c06:2]
63 unique[98000:4] = r3 + 0x18
64 *[ram]unique[98000:4] = r2
65
66 IMARK ram[6c08:2]
67 unique[98000:4] = r3 + 0x18
68 r3 = *[ram]unique[98000:4]
69
70 IMARK ram[6c18:2]
71 unique[98000:4] = r3 + 0x18
72 r2 = *[ram]unique[98000:4]
73
74
75 IMARK ram[6c1a:4]
76 unique[87700:4] = 0x20
```

```
shift_carry = CY
r2 = r2 | unique[87700:4]
tmpCY = shift_carry
tmpOV = OV
tmpNG = r3 s< 0x0
tmpZR = r3 == 0x0


IMARK ram[6c1e:2]
unique[98000:4] = r3 + 0x18
*[ram]unique[98000:4] = r2

IMARK ram[6c20:2]
unique[98000:4] = r3 + 0x18
*[ram]unique[98000:4] = r3
```

Listing 11: MMIO addresses found in function containing multiple LMS patterns in pypcode IR.

```
IMARK ram[1cc2:2]
unique[98000:4] = r3 + 0x20
*[ram]unique[98000:4] = r2

IMARK ram[1cd0:2]
unique[98000:4] = r3 + 0x24
*[ram]unique[98000:4] = r2

IMARK ram[1cde:2]
unique[98000:4] = r3 + 0x10
*[ram]unique[98000:4] = r2

IMARK ram[1cec:2]
unique[98000:4] = r3 + 0x14
*[ram]unique[98000:4] = r2

IMARK ram[1cfa:2]
unique[98000:4] = r3 + 0x18
*[ram]unique[98000:4] = r2
```

Listing 12: MMIO addresses where the base address is the same but the offset is different in pypcode IR.

```
IMARK ram[2e90:2]
unique[98000:4] = r6 + 0x0
*[ram]unique[98000:4] = r3

IMARK ram[2e92:2]
unique[98000:4] = r2 + 0x0
*[ram]unique[98000:4] = r3

IMARK ram[2e94:2]
unique[98000:4] = r5 + 0x0
*[ram]unique[98000:4] = r3

IMARK ram[2e96:2]
unique[98000:4] = r4 + 0x0
*[ram]unique[98000:4] = r3

```

```
17 IMARK ram[2e98:2]
18 unique[98000:4] = r7 + 0x0
19 *[ram]unique[98000:4] = r3
```

Listing 13: Multiple MMIO addresses that are in proximity where the register that data is send to is different, while the other register and the offset stay the same in pypcode IR.

B Ground truth dataset

Firmware name	Architecture	MCU models	MMIO segments in firmware containing addresses
atmel_6lowpan_udp_rx	ARM Cortex-M	Atmel SAMR21	0x40000000-0x42fffff
csaw_esc19_csa	ARM Cortex-M	MK20DX256VLH7	0x40000000-0x4fffff, 0xe0000000-0xe00fffff
expat_panda	ARM Cortex-M	STM Nucleo-L152RE	0x40000000-0x4fffff, 0xe0000000-0xe00fffff
nucleo_read	ARM Cortex-M	STM Nucleo-L152RE	0x40000000-0x4fffff, 0xe0000000-0xe00fffff
Nucleo_blink_led	ARM Cortex-M	STM Nucleo-L152RE	0x40000000-0x4fffff, 0xe0000000-0xe00fffff
st-plc	ARM Cortex-M	STM32F429ZI	0x40000000-0x4fffff, 0xe0000000-0xffffffff
p2im_controllino_slave	ARM Cortex-M	STM32F429ZI	0x40000000-0x4fffff, 0xe0000000-0xffffffff
stm32_tcp_echo_client	ARM Cortex-M	STM32F429ZI	0x40000000-0x4fffff, 0x50000000-0x5fffff, 0xe0000000-0xffffffff
nxp_lwip_udpecho	ARM Cortex-M	NXP FRDM-K64F	0x40000000-0x4fffff, 0xe0000000-0xe00fffff
p2im_car_controller	ARM Cortex-M	SAM3X8E	0x40000000-0x4fffff, 0xe0000000-0xe00fffff
p2im_gateway	ARM Cortex-M	STM32F103RB	0x40000000-0x4fffff, 0x50000000-0x5fffff, 0xe0000000-0xe00fffff
p2im_robot	ARM Cortex-M	STM32F103RB	0x40000000-0x4fffff, 0xe0000000-0xe00fffff
p2im_reflow_oven	ARM Cortex-M	STM32F103RB	0x40000000-0x4fffff, 0xe0000000-0xe00fffff
rf_door_lock	ARM Cortex-M	MAX32600	0x40000000-0x4fffff, 0xe0000000-0xe00fffff
sensor_oad_cc26x2lp	ARM Cortex-M	CC26x2	0x40000000-0x4fffff, 0x50000000-0x5fffff, 0xe0000000-0xe00feffc
thermostat	ARM Cortex-M	MAX32600	0x40000000-0x4fffff, 0xe0000000-0xe00fffff
zephyr-CVE-2021-3329	ARM Cortex-M	STM32L432KC	0x40000000-0x4fffff, 0x50000000-0x5fffff, 0xe0000000-0xffffffff
hello-world-arm	ARM Cortex-M	TI CC2538DK	0x40000000-0x4fffff, 0xe0000000-0xe0100000
snmp-server	ARM Cortex-M	TI CC2538DK	0x40000000-0x4fffff, 0xe0000000-0xe0100000
basic_exercises	ARM Cortex-M	LPC1549	0x40000000-0x400f0000. 0xe0000000-0xe0100000, 0x1c018000-0x1c028000
gatt_client_demo	Xtensa	ESP32	0x3ff00000-0x3ff7ffff,
ethernet_basic	Xtensa	ESP32	0x3ff00000-0x3ff7ffff,
ip_internal_network	Xtensa	ESP32	0x3ff00000-0x3ff7ffff,
scan_adv	Xtensa	ESP32	0x3ff00000-0x3ff7ffff,
synchronization	Xtensa	ESP32	0x3ff00000-0x3ff7ffff,
apsta_mode	Xtensa	ESP32	0x3ff00000-0x3ff7ffff,
ptp	Xtensa	ESP32	0x3ff00000-0x3ff7ffff,
esp_Zigbee_gateway	Xtensa	ESP32-S3	0x60000000-0x600d0fff
hello_world	Xtensa	ESP32-S3	0x60000000-0x600d0fff
proximity_polling	Xtensa	ESP32-S3	0x60000000-0x600d0fff
button	Xtensa	ESP32-S3	0x60000000-0x600d0fff
capture	Xtensa	ESP32-S3	0x60000000-0x600d0fff
peripheral_esp	Xtensa	ESP32-S3	0x60000000-0x600d0fff
ibeacon	Xtensa	ESP32-S3	0x60000000-0x600d0fff
blink	Xtensa	ESP32-S2	0x3f400000-0x3f4fffff, 0x61800000-0x61803fff, 0x60000000-0x600bffff
fuel_gauge	Xtensa	ESP32-S2	0x3f400000-0x3f4fffff, 0x61800000-0x61803fff, 0x60000000-0x600bffff
philosophers	Xtensa	ESP32-S2	0x3f400000-0x3f4fffff, 0x61800000-0x61803fff, 0x60000000-0x600bffff
minimal	Xtensa	ESP32-S2	0x3f400000-0x3f4fffff, 0x61800000-0x61803fff, 0x60000000-0x600bffff
mqtt_publisher	Xtensa	ESP32-S2	0x3f400000-0x3f4fffff, 0x61800000-0x61803fff, 0x60000000-0x600bffff
virtual	Xtensa	ESP32-S2	0x3f400000-0x3f4fffff, 0x61800000-0x61803fff, 0x60000000-0x600bffff

Table 5: Ground truth analysis samples.

C Ground truth individual sample result

Firmware name	precision	recall	f1-score
atmel_6lowpan_udp_rx	17.61	19.74	18.61
csaw_esc19_csa	26.9	14.57	18.9
sensor_oad_cc26x2lp	28.06	29.51	28.77
ip_internal_network	30.56	39.12	34.31
gatt_client_demo	28.56	45.13	34.98
st_plc	75.91	31.65	44.67
esp_zigbee_gateway	36.27	65.71	46.74
zephyr_CVE_2021_3329	67.52	36.36	47.27
p2im_robot	76.02	34.78	47.73
p2im_controllino_slave	98.74	34.85	51.52
stm32_tcp_echo_client	76.02	39.66	52.13
p2im_car_controller	100.0	35.71	52.63
blink	48.09	59.2	53.07
p2im_reflow_oven	99.2	40.38	57.4
scan_adv	59.12	60.73	59.91
p2im_gateway	95.89	43.75	60.09
rf_door_lock	75.27	51.22	60.96
expat_panda	83.5	48.94	61.71
thermostat	75.27	52.5	61.86
Nucleo_read	75.27	52.5	61.86
Nucleo_blink_led	75.27	52.5	61.86
ethernet_basic	68.34	59.49	63.61
nxp_lwip_udpecho	89.58	53.33	66.86
peripheral_esp	62.53	79.08	69.84
apsta_mode	71.02	72.18	71.6
ibeacon	66.43	79.25	72.28
virtual	65.58	81.02	72.49
snmp_server	100.0	59.09	74.28
hello_world_arm	100.0	59.09	74.28
ptp	77.96	72.18	74.96
capture	71.41	79.33	75.16
mqtt_publisher	70.68	81.02	75.5
synchronization	89.47	70.77	79.03
basic_exercises	81.36	78.79	80.05
fuel_gauge_S2	81.39	81.2	81.29
minimal	81.39	81.2	81.29
philosophers_S2	81.8	81.2	81.5
button_S3	87.59	80.28	83.78
proximity_polling_s3	87.67	80.28	83.81
hello_world	84.73	96.43	90.2

Firmware name	precision	recall	f1-score
esp_zigbee_gateway	7.14	100.0	13.33
blink	13.33	66.67	22.22
gatt_client_demo	18.18	100.0	30.77
zephyr_CVE_2021_3329	33.33	33.33	33.33
ethernet_basic	25.0	50.0	33.33
ip_internal_network	22.22	100.0	36.36
stm32_tcp_echo_client	50.0	33.33	40.0
capture	33.33	100.0	50.0
sensor_oad_cc26x2lp	50.0	66.67	57.14
p2im_car_controller	100.0	50.0	66.67
expat_panda	100.0	50.0	66.67
nxp_lwip_udpecho	100.0	50.0	66.67
p2im_gateway	66.67	66.67	66.67
atmel_6lowpan_udp_rx	50.0	100.0	66.67
basic_exercises	66.67	66.67	66.67
snmp_server	100.0	50.0	66.67
hello_world_arm	100.0	50.0	66.67
p2im_robot	66.67	66.67	66.67
hello_world	50.0	100.0	66.67
proximity_polling_s3	50.0	100.0	66.67
button_S3	50.0	100.0	66.67
peripheral_esp	50.0	100.0	66.67
ibeacon	50.0	100.0	66.67
mqtt_publisher	60.0	100.0	75.0
virtual	60.0	100.0	75.0
st_plc	66.67	100.0	80.0
rf_door_lock	66.67	100.0	80.0
csaw_esc19_csa	66.67	100.0	80.0
thermostat	66.67	100.0	80.0
Nucleo_read	66.67	100.0	80.0
Nucleo_blink_led	66.67	100.0	80.0
p2im_controllino_slave	100.0	66.67	80.0
apsta_mode	66.67	100.0	80.0
ptp	66.67	100.0	80.0
fuel_gauge_S2	75.0	100.0	85.71
philosophers_S2	75.0	100.0	85.71
minimal	75.0	100.0	85.71
p2im_reflow_oven	100.0	100.0	100.0
scan_adv	100.0	100.0	100.0
synchronization	100.0	100.0	100.0

TABLE 6: Ground truth results on all firmware samples sorted by highest average F1-score, where the left table is the address-specific category and the right the segment-specific category.

Firmware name	precision	recall	f1-score
sensor_oad_cc26x2lp	12.48	32.54	18.04
csaw_esc19_csa	27.61	14.38	18.91
atmel_6lowpan_udp_rx	21.8	30.26	25.34
p2im_robot	65.58	40.43	50.02
zephyr_CVE_2021_3329	69.23	40.52	51.12
p2im_reflow_oven	75.14	50.94	60.72
rf_door_lock	72.82	55.81	63.19
expat_panda	79.71	55.77	65.62
nxp_lwip_udpecho	86.0	53.33	65.83
stm32_tcp_echo_client	72.54	60.92	66.22
Nucleo_read	73.95	60.0	66.25
thermostat	72.87	60.87	66.33
Nucleo_blink_led	74.19	60.0	66.34
p2im_gateway	74.85	60.71	67.04
p2im_car_controller	84.09	57.14	68.04
p2im_controllino_slave	96.99	66.94	79.21
basic_exercises	81.54	80.56	81.05
st_plc	79.19	85.47	82.21
snmp_server	93.15	80.49	86.36
hello_world_arm	96.72	79.49	87.26

Firmware name	precision	recall	f1-score
stm32_tcp_echo_client	50.0	33.33	40.0
nxp_lwip_udpecho	50.0	50.0	50.0
sensor_oad_cc26x2lp	50.0	66.67	57.14
zephyr_CVE_2021_3329	50.0	66.67	57.14
p2im_car_controller	100.0	50.0	66.67
p2im_gateway	66.67	66.67	66.67
atmel_6lowpan_udp_rx	50.0	100.0	66.67
snmp_server	100.0	50.0	66.67
hello_world_arm	100.0	50.0	66.67
p2im_robot	66.67	66.67	66.67
st_plc	66.67	100.0	80.0
expat_panda	66.67	100.0	80.0
rf_door_lock	66.67	100.0	80.0
csaw_esc19_csa	66.67	100.0	80.0
thermostat	66.67	100.0	80.0
Nucleo_read	66.67	100.0	80.0
Nucleo_blink_led	66.67	100.0	80.0
p2im_controllino_slave	100.0	66.67	80.0
basic_exercises	75.0	100.0	85.71
p2im_reflow_oven	100.0	100.0	100.0

TABLE 7: Ground truth results on ARM Cortex-M firmware samples sorted by highest average F1-score, where the left table is the address-specific category and the right the segment-specific category.

Firmware name	precision	recall	f1-score
gatt_client_demo	42.14	51.62	46.4
ip_internal_network	50.95	43.53	46.95
blink	50.3	59.2	54.39
esp_zigbee_gateway	48.2	63.14	54.67
scan_adv	76.82	59.92	67.33
ethernet_basic	89.12	55.06	68.07
apsta_mode	81.91	69.17	75.0
peripheral_esp	80.12	72.38	76.05
ibeacon	81.78	72.61	76.92
ptp	87.11	69.17	77.11
virtual	79.66	78.1	78.87
capture	83.59	74.67	78.88
synchronization	94.63	68.46	79.45
mqtt_publisher	82.95	78.1	80.45
fuel_gauge_S2	89.29	78.2	83.38
minimal	89.29	78.2	83.38
philosophers_S2	89.47	78.2	83.46
proximity_polling_s3	94.03	76.06	84.1
button_S3	94.35	76.06	84.22
hello_world	93.18	92.86	93.02

Firmware name	precision	recall	f1-score
esp_zigbee_gateway	7.14	100.0	13.33
gatt_client_demo	16.67	100.0	28.58
ip_internal_network	18.18	100.0	30.77
blink	18.75	100.0	31.58
ethernet_basic	25.0	50.0	33.33
capture	33.33	100.0	50.0
hello_world	50.0	100.0	66.67
proximity_polling_s3	50.0	100.0	66.67
button_S3	50.0	100.0	66.67
peripheral_esp	50.0	100.0	66.67
ibeacon	50.0	100.0	66.67
mqtt_publisher	60.0	100.0	75.0
virtual	60.0	100.0	75.0
apsta_mode	66.67	100.0	80.0
ptp	66.67	100.0	80.0
fuel_gauge_S2	75.0	100.0	85.71
philosophers_S2	75.0	100.0	85.71
minimal	75.0	100.0	85.71
scan_adv	100.0	100.0	100.0
synchronization	100.0	100.0	100.0

TABLE 8: Ground truth results on Xtensa firmware samples sorted by highest average F1-score, where the left table is the address-specific category and the right the segment-specific category.

D False positive and negative per architecture

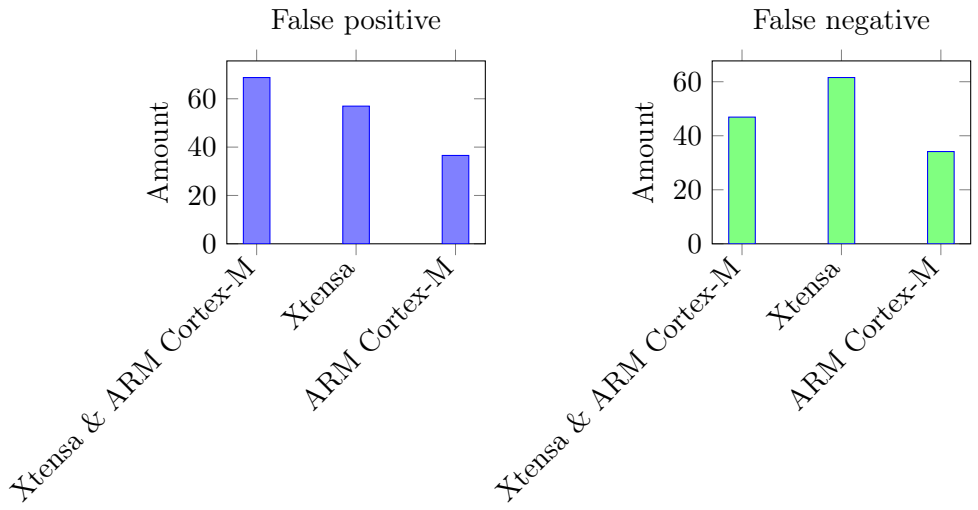


FIGURE 5: The results on the left show the average amount of unique false positives for all the architectures. The result on the right shows the average amount of unique false negatives for all the architectures.

E False positive and negative per architecture

```
    void FLASH_SetErrorCode(void)

{
  if ((uRam40023c0c & 0x10) != 0) {
    uRam20001524 = uRam20001524 | 0x10;
    uRam40023c0c = 0x10;
  }
  if ((uRam40023c0c & 0x20) != 0) {
    uRam20001524 = uRam20001524 | 8;
    uRam40023c0c = 0x20;
  }
  if ((uRam40023c0c & 0x40) != 0) {
    uRam20001524 = uRam20001524 | 4;
    uRam40023c0c = 0x40;
  }
  if ((uRam40023c0c & 0x80) != 0) {
    uRam20001524 = uRam20001524 | 2;
    uRam40023c0c = 0x80;
  }
  if ((uRam40023c0c & 0x100) != 0) {
    uRam20001524 = uRam20001524 | 1;
    uRam40023c0c = 0x100;
  }
  if ((uRam40023c0c & 2) != 0) {
    uRam20001524 = uRam20001524 | 0x20;
    uRam40023c0c = 2;
  }
  return;
}
```

LISTING 14: Code snippet showing that the LMS pattern also can be used with RAM instead of MMIO. The uRam200x address symbol name in the ELF is pFlash.ErrorCode.