FIRMLINE: a Generic Pipeline for Large-Scale Analysis of Non-Linux Firmware

Alexander Balgavy[†]
Independent
alex@balgavy.eu

Marius Muench[†]
University of Birmingham
m.muench@bham.ac.uk

Abstract—**Embedded devices are a pervasive and at times invisible part of our lives. Due to this pervasiveness, security vulnerabilities may have severe consequences, particularly because many embedded devices are deployed in sensitive applications, such as the industrial, automotive, and medical sectors. Linux-based firmware has already been the subject of extensive research; however, a considerable part of embedded devices do not run Linux. Since current literature mostly focuses on Linux-based firmware, the ecosystem of non-Linux firmware is not well-known.**

Therefore, in this paper, we aim to fill this gap in research with FIRMLINE, a pipeline suitable for a large-scale study of non-Linux-based firmware. Using this pipeline, we analyze 21,755 samples, obtained from previous studies and new sources. As part of a security assessment, we also investigate the presence of operating systems and memory protections for a subset of 3262 non-Linux ARM samples and find that the majority do not make use of either. Our work will allow for further research of non-Linux firmware, such as refining generic analysis techniques or investigating the OS and deployed security facilities of such firmware in more detail.

I. INTRODUCTION

Embedded devices are an ever more ubiquitous part of our lives, and especially the rise of the so called 'Internet of Things' (*IoT*) led to an unforeseen growth of networked embedded devices. Although there is no precise data for the number of devices currently in use, projections frequently estimate more than 15 billion active IoT devices for 2023 [1, 2].

Since these devices are so wide-spread, their security is of the essence, especially if the devices fulfill a life-critical function. However, embedded systems may have security vulnerabilities, just like ordinary desktop software. Numerous attacks targeting these vulnerabilities demonstrate the severe consequences of insecure embedded devices. For example, the MIRAI botnet [3, 4], which compromised mainly IoT devices, had a peak infection count of 600k and was used for distributed denial-of-service (DDoS) attacks causing multiple-hour outages at large companies (including Twitter, Netflix, Reddit, and GitHub). In fact, Mirai variations and descendants are prominent up to this day and infections are still observed on internet-connected devices [5, 6]. Considering the lasting prominence of vulnerabilities in embedded devices, and their potentially devastating effects, it is necessary to study the security of these devices and their associated *firmware*.

[†]Contributions were partly made while at Vrije Universiteit Amsterdam.

Workshop on Binary Analysis Research (BAR) 2024
1 March 2024, San Diego, CA, USA
ISBN 979-8-9894372-0-7
https://dx.doi.org/10.14722/bar.2024.23008
www.ndss-symposium.org

Previous work on firmware analysis frequently focus mainly on Linux-based firmware (e.g., [7, 8, 9, 10, 11, 12, 13, 14, 15, 16]). This is particularly true for large-scale analyses, most of which have been carried out on Linux-based firmware [7, 8, 9, 10, 11]. Despite recent advances in *rehosting* [17, 18], most studies on non-Linux-based firmware narrow their scope to a few vendors and/or a specific architecture and incorporate domain knowledge. However, this knowledge cannot be generalized to unknown non-Linux-based firmware, and we argue that generalization is necessary to account for the diversity of embedded devices. Hence, given the lack of generic analysis techniques, the ecosystem of a large portion of firmware (i.e., that which is not based on Linux) is unknown.

Therefore, in this paper we set out to create a platform to enable a large-scale security analysis of non-Linux firmware. In particular, we present FIRMLINE, an open-source analysis pipeline for unknown firmware. To support firmware targeting unknown devices, we attempt to develop generic approaches that do not rely on prior knowledge about the target device's characteristics (e.g., the specifications of its CPU). We apply heuristics to learn high-level facts about the firmware, and use those facts to inform later stages of the analysis. These later stages are necessarily architecture-specific (e.g., code analysis), but the techniques themselves are generic and can be adapted for other architectures.

To obtain enough non-Linux firmware for a large-scale study, we re-use existing datasets of such firmware; we also download additional firmware, which we filter as an early step in our analysis pipeline.

In summary, we:

- create and analyze a large-scale data set comprising of 21,755 unique firmware samples, collected from six publicly-available sources.
- develop FIRMLINE, an open source and modular pipeline for analysis of non-Linux firmware.
- investigate the presence of security critical features for 3262 arm based samples.

Our dataset, analysis pipeline, and analysis results are publicly available at https://gitlab.com/firmline/firmline.

II. Background

A. Firmware architecture and formats

Firmware is the software that is stored on, and controls, an embedded device. However, the landscape of firmware is not uniform and used operating systems, instruction set architectures, and file formats are by far more diverse than for software targeting desktop systems.

Operating systems. A large number of firmware targets devices running a variant of the Linux Operating System (*OS*), which we will simply refer to as Linux-based. However, an embedded device need not use Linux; we refer to the programs controlling such devices as non-Linux-based firmware. The latter may be targeted at a device that runs a real-time or other standard OS, a Windows OS, a custom OS made by the manufacturer, or that does not have an OS abstraction at all.

Instruction set architectures. Embedded devices deploy a variety of instruction set architectures and variants. The ARM architecture [19] is among the most prevalent and according firmware is commonly analyzed in the literature, while other widely used architectures, such as MIPS [20], received less attention. The ARM architecture has multiple versions, such ARMv7 for 32-bit devices or ARMv8 which introduced support for 64-bit devices. Despite the differences in versions, modern ARM architectures also have different different 'profiles', which provide different functionality in terms of instructions and architectural components. Among embedded devices, the R and M profiles are most prevalent.

File Formats. There is no standard on how to store and distribute firmware, and in practice manufacturers commonly choose their own format. Firmware may be compressed using known compression schemes, it may be encrypted, or it may be packed with a packer program to thwart reverse-engineering. A common format for firmware is Intel HEX (*IHEX*), which represents the binary object in ASCII [21] and includes some additional metadata such as the address of the individual chunks of code and data. However, firmware may also be distributed as a raw binary file, with a custom header or with no header at all.

B. Binary Program Analysis

When analyzing firmware, access to source code is the exception, which is why we need to rely on binary analysis techniques. Commonly, these are distinguished in three approaches: dynamic, hybrid, and static.

Dynamic Analysis requires to executed the analyzed firmware. One option is black-box analysis, using the actual device that the firmware targets. Another option is execution in a virtual environment, ideally as close as possible to how it would execute in real-world use. This can be done through emulation, which faithfully recreates the full hardware model virtually including peripherals, or rehosting, in which either only the necessary parts of hardware are inferred, or peripheral interaction is forwarded directly to the physical device [17].

Hybrid Analysis. In hybrid forms of analysis, such as symbolic execution, the analysis engine models all system state (registers, memory, etc.) virtually, and 'pretends' to execute parts of the code by acting out operations on this virtual state. Instead of concrete values, a symbolic execution engine uses symbols, which are bound by mathematical constraints on their possible values; the engine then uses an SMT solver to obtain concrete values satisfying those constraints.

Static Analysis. In static analysis, the goal is to understand program behavior and properties only by inspection, without executing its code. To this end, analysis engines try to transform a binary program into a more accessible format, such as assembly instructions, intermediate representations (IR), or pseudocode. Once transformed, they can analyze program flow using, for example, control flow graphs (*CFG*s): graphs where nodes represent basic blocks (series of instructions with one entry and exit point), and edges indicate the possible control flow between basic blocks. Another example option is data flow analysis: tracking which data definitions can reach a certain point in the program, for example to identify paths for attacker-controlled data.

Firmware Specific Challenges. In the context of firmware, program analysis presents more challenges, particularly if nothing is known about the binary in question. Even basic static analysis involves several preliminary steps. Assembly instructions and their operands are represented by specific sequences of bytes, and the interpretation of the sequences differs by architecture. Hence, disassembling or lifting a binary depends on the architecture, so identifying the architecture is the first step. Next, there is the question of where in memory the firmware is located and where to begin disassembly: since instructions and operands are sequences of bytes, starting from a different offset in the binary will result in some operands being interpreted as instructions, and vice-versa – yielding incorrect code. Similarly, indirect jumps can only be correctly resolved if the firmware code is mapped at the right location. Therefore, it is necessary to identify the proper base address for the firmware and the proper entry point for disassembly, and to use it throughout the rest of the analysis. With this information, one can disassemble a binary, inspect the assembly code to understand control and data flow, and perhaps even decompile the assembly into pseudocode.

III. Data Collection

Scope. In this study, we focus on low-level non-Linux firmware. We assume that this type of firmware is mainly distributed as binary blobs which are neither encrypted nor compressed. Hence, our data collection approach (and later analysis passes) focus on single binaries assumed to be loaded in a single flat address space.

Methodology. We create the data set for this study using two groups of sources: already existing collections of firmware that we downloaded manually, and individual samples that we scraped. The majority of samples come from already existing datasets: FirmXRay's dataset [22, 23], the Linux firmware repository (tag `20230210`)[*] [24], the Roccat firmware collection [25], the Monolithic Firmware Collection (commit `a2458fe`) [26], and the Tadiphone repository group [27].

While some of these datasets were subject to prior academic studies, some of them have, to the best of our knowl-

[*]Note that, despite its name, this repository does not hold Linux-based firmware, but firmware blobs to be used with the Linux kernel.

Data source	Previous studies	#Samples
FirmXRay	HEAPSTER [28], FIRMXRAY [22]	790
Linux Firmware	Iooss and Campana [29] (1 sample)	2194
Roccat	None	27
Android dumps (Tadiphone)	Hou et al. [30]	18315
Monolithic Firmware Collection	None (as a whole)	128
IHEX files from GitHub	None	301

Table I: Datasets of this paper, their use in previous studies, and unique number of samples after deduplication.

edge, not been analyzed in previous work. Table I shows which past studies analyzed the datasets that we include in this paper. The Linux firmware dataset contains firmware for use with the Linux kernel, targeted at devices such as WiFi cards, graphics cards, and devices for digital signal processing. The Tadiphone group of repositories contains firmware dumps from Android devices, including phones such as Aligator S6500 and Infinix Smart 4 Plus. The Monolithic Firmware Collection contains images of monolithic firmware, collected from previous studies; while the samples themselves have been used in the individual papers (e.g., FIRMXRAY), the collection as a whole has not yet been used. Finally, the Roccat repository contains firmware for Roccat devices like keyboards and mice.

While most data sources did not require special processing and we could retrieve the firmware directly, some automation was needed for the Tadiphone firmware. The Tadiphone repository group contains a large number of repositories, and not all of them contain firmware. Therefore, we wrote a scraper utilizing GitLab's API to search for repositories in the group which contain known firmware paths. We then cloned the repositories and imported firmware files from those known paths.

Lastly, we also scraped IHEX files from GitHub. We initially tried to use GitHub's search API, but the results were unreliable (often, the API would return a number indicating there were results found, but the actual list of the results was empty) and we frequently encountered rate limiting. Therefore, we use Selenium [31] in combination with geckodriver [32] to access GitHub's advanced search page to find repositories containing files with the IHEX extension. Using GitHub's search operators, we excluded already known repositories, at least to the degree that we could, considering GitHub's 500 character search limit. While we needed to trigger additional reloads to overcome reliability issues, this process yielded a list of repositories known to contain files in IHEX format which we could then clone.

Dataset Composition. Our final dataset contains 21,755 samples; we filtered out 34,001 duplicate files, with 8394 unique hashes. By source, the duplicate hashes matched with 27,469 samples from the Tadiphone repositories, with 5580 from the Linux firmware repository, with 150 IHEX files from GitHub, with 4 from the Monolithic Firmware Collection, and with 798 from FIRMXRAY. We show the composition of the deduplicated dataset in Table I.

IV. DESIGN

We now present FIRMLINE, our pipeline for analyzing firmware samples. Figure 1 shows an overview of FIRMLINE and its different analysis components.

A. Overview

When a new firmware sample is processed, we first establish whether it is a duplicate or already-processed file (1). If this is not the case, we start a preliminary analysis, collecting simple characteristics (2). Then, the file passes through two stages where we attempt to detect information about the firmware's target device: whether the target is Linux-based (3), and the target's architecture (4). The next analysis stage detects the base address of the binary and starts disassembly of the binary (5). The last analysis step carries out analyst defined code analysis passes (6). Finally, we persist the results of these analyses in a database (7). Analysis stages may time out, in which case we store information about this timeout in the database. We also note that, step 5 and 6 require architecture specific knowledge and are, thus, not fully generic. However, the modular design of our pipeline allows for easy modification and integration of different implementations for different architectures. The remainder of this section covers the individual analysis stages in more detail.

B. Preprocessing (1)

When a new sample is processed, if it is an IHEX file, we first convert it to a binary file. Next, we calculate a hash over the binary to detect duplicates and already processed files. We check whether we had already processed this file before by comparing hashes; if they match, then we also carry out a byte-wise comparison of file contents to detect potential hash collisions.

C. Preliminary Analysis (2) and Linux Detection (3)

Once we have determined that we have not yet processed a given file, we collect some surface-level properties (step 2 of Figure 1): the size of the file, its entropy, and whether it contains strings of at least 32 consecutive `0xFF` or `0x00` bytes, which are often used as padding in binary firmware. We check for this padding because its presence may indicate that the file in question is an executable, rather than plain data. In step 3 of Figure 1, we scan the file for known signatures; if it contains contains strings known to be present in Linux-based software, (i.e., any of 'uimage' or 'u-boot', 'squashfs' [33], 'linux'), or a string denoting a filesystem, we consider the firmware target to be Linux-based.

D. Architecture Detection (4)

The next stage of FIRMLINE detects the architecture of the firmware's target device (step 4 in Figure 1). It consists of two parts: scanning and verification. In the first step, we follow the insights of Granboulan [34] and scan a binary using statistical analysis techniques to determine possible architecture candidates. In the second step, we validate the architecture candidates with a reverse engineering framework: if the framework identifies functions, we consider the detected architecture to be correct. To reduce analysis time, we only

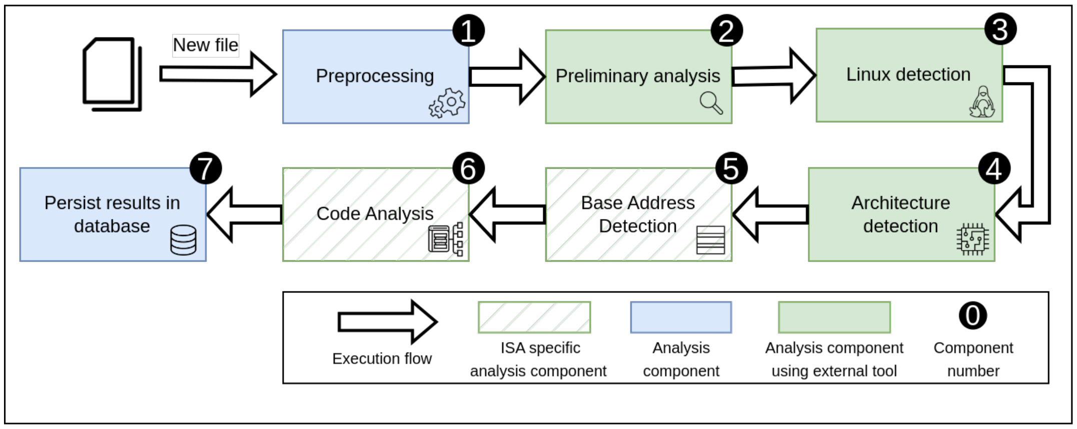

Figure 1: FIRMLINE analysis pipeline.

test for the top two most probable architectures, and select the one with the highest number of detected functions. To find functions, reverse engineering frameworks may apply a wider range of techniques outside the scope of this study, such as function prelude search, call analysis, reference analysis, emulation to find computed references, type matching, non-return analysis, or graph reference analysis [35]. Many architectures support little-endian, big-endian, or a combination of the two; in those cases, we try all possibilities, and choose the one with the highest number of detected functions as the correct architecture.

E. Base address detection and disassembly (5)

This stage disassembles the binary using information obtained in the previous stages. Before a binary can be disassembled, it must be loaded at the correct base address, otherwise the disassembly will be wrong. For firmware formats where the base address is unknown, we deploy heuristics to resolve it. We adopt an approach similar to the state-of-the-art, FIRMXRAY [22], with some modifications.

The main insight behind FIRMXRAY's approach is that firmware will require absolute pointers during runtime. Hence, FIRMXRAY first identifies multiple potential absolute pointers to resolve the base address for ARM binaries: (1) absolute function pointers used for indirect branches, (2) absolute pointers to strings passed as arguments to functions, and (3) vector table entries. Based on the detected absolute pointers, FIRMXRAY models different base addresses and chooses the candidates which allows most pointers to resolve correctly. However, we note that absolute string pointers and vector table entry pointers require domain knowledge: FIRMXRAY uses information from vendor-specific SDKs to identify which functions accept string pointers as parameters, and knowledge about the memory layout of IoT devices to locate the vector table. Since absolute function pointers are the only type that can be readily generalized, this is what we implement in FIRMLINE, and what we will focus on for the remainder of this section.

Our approach, following FIRMXRAY's absolute function pointer detection and base address identification, consists of four steps:

1) Disassemble the binary from offset 0,
2) Scan the disassembly for absolute function pointers,
3) Scan the disassembly for function entry points,
4) Try all possible offsets and select the offset with the most absolute pointers resolving to valid function entry points.

In this process, the detection of absolute function pointers and function entry pointers require architecture-specific knowledge. Indirect branch targets can either be encoded via registers, memory, or both, dependent on the target architecture. While indirect branches via memory may directly encode the branch target as an absolute pointer, we apply simple backwards taint analysis to identify the target for indirect branches via registers. Furthermore, one prevalent technique for function entry detection scans for instructions usually emitted by compilers for function prologues. This necessarily requires knowledge of the target architecture and, thus, cannot be implemented in an architecture-independent way.

Once we detected the absolute function pointers and function entry points, we try to find an offset that maximizes the number of function pointers that match with function prologues. We iterate over integers in the range from 0 to the smallest address in the list of absolute pointers; we know the upper bound because loading the binary at an address above the smallest function pointer would render that pointer invalid. For each number (offset) x in this range, we calculate for each absolute pointer p the difference $d = p - x$ (i.e., we reduce the address of the pointer by the offset). If the offset x is correct, this altered pointer d should be in the list of function prologues. We choose the final offset depending on how many function pointers, modified with that offset, resolve to valid function prologues. This is also why false positives in function entry or branch target detection do not impact accuracy: the constraint solving approach will reduce the set of possible functions to only those with valid references (though a very high number of false positives may impact processing speed).

F. Code Analysis (6)

Once we have calculated a base address, we rebase the binary to that address, and disassemble it again. Now, different code analysis passes can obtain additional knowledge about the target firmware. For instance, one pass could detect the presence of security critical features, while others aim at the detection of string-processing functions. Overall, the code analysis stage is meant to be extended by analysts to extract desired knowledge about a specific subset of firmware samples.

G. Data Storage (7)

Once the previous stages are completed, we store the analysis results in a relational database, using a cryptographic checksum of a file as its unique identifier. We choose a relational database, because it allows us to encode consistency checks into the schema. For example, if an architecture is not detected, there will be no base address detection and analysis data, but if one is detected, the presence of data depends on whether the architecture is supported by the analysis system. These checks can be validated at insertion time, ensuring that all data we persist is consistent.

V. IMPLEMENTATION

We implemented a proof of concept of FIRMLINE in Python 3 and POSIX shell scripts, with 1169 custom SLOC, as measured by Tokei [36]. This code connects the various tools, collects results, and saves them to a database. We persist data in an SQLite database, with constraints encoded into the schema to ensure consistency. The user can change configuration parameters for the pipeline by modifying a configuration file, following the INI file structure. Below, we outline the implementation specific details for the individual analysis stages.

Preprocessing and Preliminary Analysis. For conversion of IHEX files, we use Bincopy [37], which can convert such files to (among others) a binary format. To search for padding in the file, we use Bgrep [38]. We then employ Binwalk [39] for an entropy scan, and we run its signature scan option to detect Linux-based firmware in step 3 of Figure 1.

Architecture Detection. To detect the architecture in step 4, we use CPU_REC [34]. Since its Python API only outputs one result, we modify its code to return the top two architectures. We pass the architectures detected by cpu_rec to Radare2 using r2Pipe, and run two code analysis commands: 'aaaa' (auto analysis) and 'aab' (basic-block analysis using the Nucleus [40] algorithm). If Radare2 finds functions, we consider the file to indeed be firmware intended for the architecture detected by cpu_rec. Some architectures are supported by more than one disassembler in Radare2 (e.g. for SPARC, Radare2 has both `sparc` and `sparc.gnu`); in those cases, we try all disassemblers that support a given architecture.

Base Address Detection. For base address detection, we use Ghidra v10.3 [41] in headless mode, with a Python script to automate analysis. To load a file, Ghidra requires us to be explicit about the language of the file, specifying the following: the architecture, whether the file is big-endian or little-endian, whether it is 32-bit or 64-bit, and what variant of the architecture it targets. However, we do not get this level of detail from cpu_rec, or any other similar tool with comparable reliability, so we need to choose a language that would offer the widest support.

In our proof of concept, we decide to implement our base address detection for ARM binaries: ARM is the market leader in many domains of embedded devices [42, 43, 44, 45], so this choice is in line with previous work. For ARM, previous steps of the pipeline narrow down the instruction set architecture to one of the following cpu_rec defined strings: ARMhf, ARMel, ARMeb, ARM64. We decide to load ARMhf and ARMel as ARM 32-bit Cortex little-endian, and ARMeb as ARM 32-bit Cortex big-endian, because the Cortex variants offer a superset of possible features (and if a particular sample does not target the Cortex variant, it simply will not use those extra features). For ARM64, we try both the 64-bit and 32-bit variants, and select the one with the highest number of cross-references in Ghidra. If a binary uses Thumb instructions, Ghidra can automatically detect this, and interpret the bytes as Thumb where necessary.

When looking for absolute function pointers, we search for branch instructions, such as `blx`. Those instructions branch to a memory address stored in the operand register, so unless the operand is the link register, we use simple backwards taint analysis to identify the load populating the branch destination register. To detect function entries, we scan for instructions typically encountered in function prologues on ARM to set up the function's stackframe (e.g., the `push`-pseudo instruction or `stmfd`). We also query Ghidra for addresses of code blocks that it identified as functions, and add them to the list of function prologues. Ghidra uses additional techniques to find functions: it decompiles code to determine unknown calling conventions, searches for architecture-specific byte patterns, discovers non-returning functions, and creates function references for code that is called like a function [46].

Code Analysis. In our proof-of-concept implementation of FIRMLINE, we implement two security analysis passes with Ghidra: detection of presence of OS abstractions, and whether the firmware accesses a Memory Protection Unit (MPU).

To identify OS abstractions, we leverage an intrinsic of the ARM instruction set: the *SuperVisor Call* (or, in short, the `svc` instruction). This instruction typically denotes a call to an operating system to provide a service [47]. Therefore, we search the code for instances of the `svc` instruction, and if we find it, we infer that the firmware likely uses an OS abstraction.

Furthermore, memory-level protection may enhance the security of firmware, and ARM-based embedded systems may deploy an MPU. The MPU is a hardware unit that allows software to define and change attributes and access permissions for memory regions. On ARM devices, the MPU can be configured in two ways: M-profiles usually access the memory-mapped MPU configuration registers [48], while R-profiles can interface with the system co-processor for MPU configuration via specialized instructions [49]. Therefore, we search the firmware for these two behaviors, and if we find either of them, we conclude that the program interacts with an MPU.

Timeouts. Parts of the pipeline and analysis passes may take a long time or not terminate at all. To ensure operation throughout a large-scale analysis, we add timeouts to the callbacks into different tools: 5 minutes for entropy calculation via binwalk, 10 minutes for a signature scan with binwalk, 30 minutes for function detection with Radare2, and 30 minutes for base address detection and code analysis with Ghidra. The timeouts are configurable by the user, through the central INI configuration file of FIRMLINE.

VI. Evaluation

We evaluate FirmLine on a server running Ubuntu 22.04.1 LTS, with 4 single-thread CPU cores of an AMD EPYC 7662 processor and 8 GB of RAM; we use 13 GB of storage.

We first determine the correctness of our base address calculation (subsection VI-A), and then evaluate FirmLine on our collection of firmware samples. We first try to answer several questions about our data set: how many samples target non-Linux firmware, what architectures are targeted, and what proportion of firmware samples have a non-zero base address (subsection VI-B). Afterwards, we assess the presence of operating systems and MPU configurations using our proof-of-concept analysis passes (subsection VI-C). Lastly, we report key performance metrics for FirmLine, such as analysis time and number of timeouts (subsection VI-D).

A. Accuracy of Base Address Detection

To evaluate FirmLine's base address detection capabilities, we compare against FirmXRay as the state of the art and consider its results as ground truth. Therefore, we run FirmXRay on its dataset (which is a subset of our dataset), and compare its base address results with the results of FirmLine for the same dataset. As an additional comparison point, we also run FirmXRay while disabling the components that require domain knowledge about the target device; we will hereafter refer to this modified version as *FirmXRay-M*.

In total, FirmXRay's dataset contains 790 samples. FirmXRay processed 780 of them, while FirmXRay-M processed all 790, and the approaches resolve a matching base address for 603 samples. In contrast, FirmLine only resolved a base address for 307 samples. In more detail, all samples had an architecture detected, and 1 sample's architecture was detected as IA-64, which is not supported by Radare2. For the rest of the samples, 10 target the `6502` architecture, and 779 target ARM. In the Radare2 analysis, which should confirm or refute the detected architecture, 45 of the samples timed out, all of them ARM. 744 were analyzed, of which 10 did not have functions detected, and the remaining 734 did. Those 10 were all reported as targeting `6502` by cpu_rec, but the analysis with Radare2 correctly refuted this (by not detecting functions), because all of the samples in FirmXRay's dataset are known to target ARM [22]. Therefore, our system successfully detected the architecture of 734 samples – all of them targeting ARM, as expected. While resolving the base address, 422 failed because of an error: 207 because FirmLine could not detect absolute function pointers, 214 because an offset satisfying the constraints could not be found, and 1 could not be disassembled by Ghidra.

Out of the 307 successfully analyzed samples, FirmLine resolved the same base address as FirmXRay for 3 samples, and the same address as both variants of FirmXRay for 40 samples, for both zero and non-zero offsets. Our results indicate that our approach, while not relying on domain specific knowledge, only resolves the correct base address for a small fraction of samples. Therefore, further development is is needed to improve FirmLine's success rate.

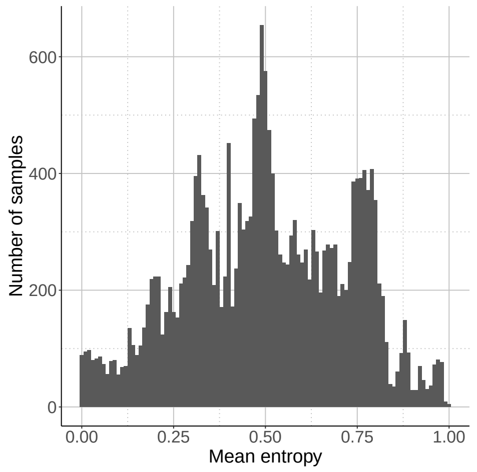

Figure 2: Mean entropy of 21,755 samples.

B. Landscape of Non-Linux-Based Firmware

To assess the landscape of non-Linux-based firmware, we follow the data set through the different analysis stages of FirmLine. We further summarize the analysis results after filtering out Linux samples in Table II.

Preanalysis. We first analyze the entropy of all 21,755 samples in our dataset. As Binwalk outputs an entropy value for each detected section of the file, we calculate the mean across the whole file. We choose the mean as a metric because the values can vary significantly, and we want extreme values to have a higher effect on the result. For each sample, we plot the mean entropy in Figure 2. The majority of samples have an entropy below 0.7, which suggests that they are likely not encrypted, packed, or compressed, confirming our assumption noted in Section III. When searching for padding bytes, 116 samples contained at least 32 consecutive `0xFF` bytes, 8281 contained at least 32 consecutive `0x00` bytes, 6015 contained both, and 7343 contained neither.

Linux detection. Based on output from Binwalk, we detected 248 samples as Linux-based (around 1.16% of the total dataset). Of this 248 samples, 185 were detected via signature matching and 63 via included file systems. In terms of data source, 19 were from the Linux firmware repository, 1 from the Roccat repository, and 228 from the Tadiphone dumps. These results are also summarized in Table III.

Architecture detection. Out of the remaining 21,507 non-Linux samples, FirmLine detected an architecture for 16,938 samples via cpu_rec. Table IV shows the top architectures as reported by cpu_rec and how many of them were confirmed with Radare2. For these samples, 9475 had an architecture supported by Radare2. FirmLine could not confirm the architecture for 1101, as Radare2 timed out given the limit specified in section VI. The remaining 8221 were analyzed successfully with Radare2, and the architecture as suggested by cpu_rec was confirmed for 3297 of them. Interestingly, in none of these cases `6502` was confirmed as the architecture by Radare2, indicating a high false positive rate for cpu_rec (at least for this specific architecture), and the importance of FirmLine's architecture verification step.

Source	Architectures	Verified	Base 0	Base not 0	MPU usage	`svc` instructions	Avg. analysis time
FirmXRay	ARMhf (779), 6502 (10), IA-64 (1)	734	22	285	0 / 0	287	00:09:18
Linux Firmware	6502 (207), None (198), Xtensa (159), IA-64 (157), ARcompact (156), ARMhf (150), TMS320C6x (126), 8051 (108), X86 (81), ARMel (78), NDS32 (77), OCaml (74), i860 (65), NIOS-II (59), MMIX (57), IQ2000 (49), Mico32 (39), Cray (36), MIPSeb (30), STM8 (27), MIPS16 (22), MN10300 (20), VAX (19), 68HC11 (18), RISC-V (16), TriMedia (15), MSP430 (15), Epiphany (12), SPARC (11), MIPSel (11), HP-Focus (10), SuperH (8), Blackfin (8), WE32000 (7), FT32 (6), Cell-SPU (4), Visium (4), AxisCris (4), ROMP (3), WASM (3), AVR (3), V850 (3), ARC32eb (2), #6502#cc65 (2), Alpha (2), RX (2), PPCeb (2), ARM64 (1), FR-V (1), ARMeb (1), Stormy16 (1), PIC10 (1), CUDA (1), PDP-11 (1), M88k (1), M68k (1), ARC32el (1)	550	7	38	0 / 1	23	00:07:07
Roccat	ARMhf (23), SuperH (1), None (1), 6502 (1)	23	0	0	0 / 0	0	00:06:20
IHEX (GitHub)	ARMhf (79), NDS32 (60), Stormy16 (59), 6502 (46), 8051 (29), MMIX (7), None (6), Xtensa (3), MN10300 (2), i860 (2), ARcompact (1), Visium (1), RL78 (1), ARMel (1), OCaml (1), FR30 (1), CompactRISC (1), IQ2000 (1)	104	65	0	0 / 2	4	00:07:19
Android dumps (Tadiphone)	6502 (4435), None (4362), MMIX (1472), OCaml (1224), ARMhf (973), NDS32 (852), IA-64 (778), Xtensa (485), STM8 (299), ARM64 (284), Moxie (271), IQ2000 (255), WASM (235), ARMel (211), SuperH (182), i860 (140), TMS320C6x (109), Visium (107), VAX (103), 8051 (98), ARcompact (94), MIPS16 (90), FR-V (90), Cray (77), Stormy16 (71), MIPSel (65), MIPSeb (59), Z80 (44), HP-Focus (42), CLIPPER (41), RISC-V (39), Blackfin (36), MSP430 (35), S-390 (34), PIC10 (32), PIC16 (32), Epiphany (32), TriMedia (31), ARC32el (19), MicroBlaze (18), CompactRISC (17), MN10300 (15), X86-64 (14), FT32 (13), Mico32 (12), PIC24 (12), PDP-11 (12), NIOS-II (11), WE32000 (11), ARMeb (11), M32R (11), RX (9), ROMP (9), ARC32eb (8), M68k (8), H8S (7), PPCeb (7), FR30 (6), 68HC11 (5), AxisCris (5), MCore (5), AVR (4), SPARC (3), Alpha (3), X86 (2), M88k (2), V850 (2), PIC18 (2), TLCS-90 (2), H8-300 (1), Cell-SPU (1), CUDA (1)	1800	59	102	0 / 24	43	00:03:57
Monolithic Firmware Collection	ARMhf (119), ARMel (3), None (2), MMIX (1), OCaml (1), IA-64 (1), 6502 (1)	76	28	6	0 / 1	1	00:04:36

Table II: Summary of analysis results for 21,507 non-Linux samples in our dataset (analysis time in hours, minutes, and seconds).

Source	Padding	No padding	Linux
FirmXRay	0x00 (542), 0xFF (8), Both (68)	172	0
Linux Firmware	0x00 (1419), 0xFF (6), Both (266)	503	signature (15), fs(4)
Roccat	0x00 (23), 0xFF (1), Both (1)	2	fs(1)
IHEX (GitHub)	0x00 (135), 0xFF (13), Both (72)	81	0
Android dumps (Tadiphone)	0x00 (6076), 0xFF (88), Both (5568)	6583	signature (170), fs(58)
Monolithic Firmware Collection	0x00 (86), 0xFF (0), Both (40)	2	0

Table III: Summary of preliminary analysis results.

Architecture	Count	#Confirmed by r2?
6502	4758	0
ARMhf	2197	1605
MMIX	1538	-
OCaml	1323	-
NDS32	989	-
IA-64	939	-
Xtensa	648	276
STM8	327	-
IQ2000	314	-
ARMel	296	143
ARM64	295	266
Moxie	271	-
ARcompact	251	244
WASM	238	-
8051	236	199
TMS320C6x	235	-
i860	207	-
SuperH	191	183
Stormy16	131	-
VAX	122	125
Other	1780	257

Table IV: Number of samples detected by cpu_rec and confirmed by Radare2 for the 20 most common detected architectures. Other architectures are grouped under *"other"* and "-"indicates that the architecture is not supported by Radare2.

Base Address Detection. From the 3297 confirmed samples, 2019 had an architecture supported by FIRMLINE's Ghidra analysis. While detecting the base address, 22 samples timed out in the initial loading stage. For the rest, 1391 failed to resolve a base address: 1068 because of a lack of absolute function pointers, 1 because of a lack of function prologues, and 322 because an offset satisfying the required constraints could not be found for the sample. While some samples had a zero base offset, the majority had a non-zero offset, as shown in Table II.

This leaves a total of 606 samples which made it through the pipeline and are ready for the analysis passes: 307 from FIRMXRAY, 65 IHEX files from GitHub, 39 from the Linux firmware repository, 34 from the Monolithic Firmware Collection, and 161 from the Tadiphone firmware dump repositories.

C. Security Assessment of Non-Linux-Based Firmware

Next, we analyze the firmware for two properties with potential security implications: interaction with an operating system and use of memory protections. For this part of the evaluation, we do not consider only the 606 samples as established in the last subsection, but also consider a wider data set. In particular, we amend the data set with samples that are, according to cpu_rec, supported by our analysis passes, but which did not pass through the architecture verification step with Radare2. This expanded data set consists of 3262 samples. Out of these samples, FIRMLINE could resolve the base address and continue to the security analysis for 771.

OS Access/Usage. Our first analysis pass aims to infer the presence of an OS by checking whether the firmware contains `svc` instructions. Out of the 606 samples that made it through the pipeline, 357 contain such instructions, the majority at more than 10 distinct locations. These 357 come from three sources: 287 from FIRMXRAY, 4 from IHEX files, 22 from Linux firmware, 1 from the Monolithic Firmware Collection, and 43 from the Tadiphone group. For the expanded data set we reach a total of 415 out of 771 samples with `svc` instructions. The majority of samples still have such instructions at more than 10 distinct locations each. In terms of sources: 293

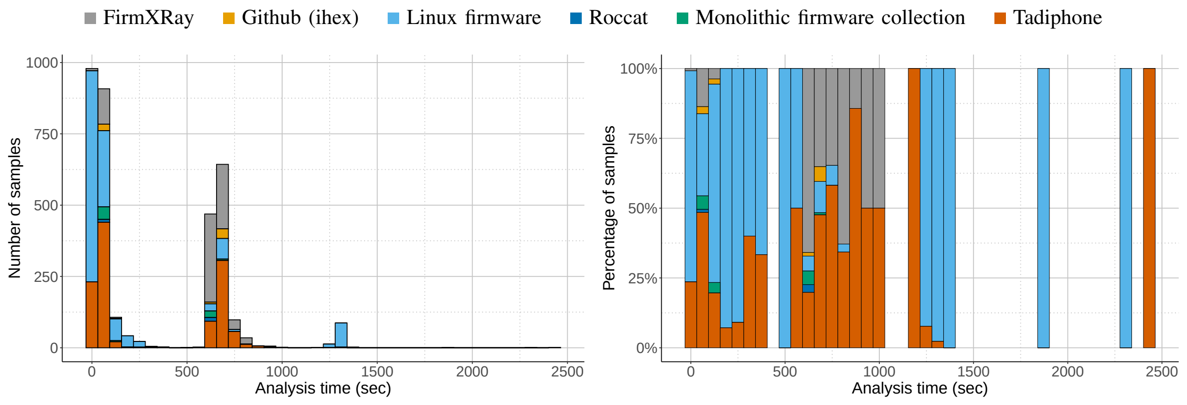

Figure 3: Analysis time for successfully processed samples.

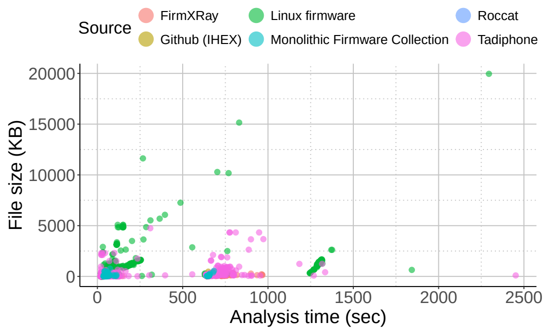

Figure 4: File size vs full pipeline analysis time for successfully processed samples.

samples are from FIRMXRAY, 4 from IHEX files, 39 from the Linux firmware repository, 5 from the Monolithic Firmware Collection, and 74 from the Tadiphone group.

We further conduct a manual analysis of 30 randomly selected samples: 15 with detected OS access, and 15 without. In this manual review, we verified the communication (or lack of communication) with an OS through Ghidra's graphical interface, by checking for `svc` instructions. In addition, we also searched for the string "svc" in the binary, using Ghidra's string search. The results are in Table VI in Appendix A. For 27 samples, our manual analysis found the same results as FIRMLINE. However, in 3 samples, our string analysis found strings that indicate the presence of an OS, such as "Unknown SVC 0x%02lX called at 0x%08lX", "Tsk_Ssvc", and "Elf_Ssvc". We cannot know for certain, but if the samples with these strings are targeting a device with an OS abstraction, FIRMLINE would have a false negative rate of 16.7%.

MPU Access/Usage. Applying our approach to 606 samples that passed through the pipeline, we found 28 samples that access memory within the address range designated for the MPU: 2 in converted IHEX files, 1 in the Linux firmware repository, 1 in the Monolithic Firmware Collection, and 24 in the Tadiphone firmware repository group. None of the samples used dedicated instructions to access the MPU. The extended dataset yielded 14 more samples accessing the MPU memory range: 1 from the Linux firmware repository, 12 from the Monolithic Firmware Collection, and 1 from the Tadiphone dumps. However, even in this larger dataset, there were no samples that used dedicated instructions for MPU access.

To check FIRMLINE for false positives/negatives, we conducted a manual analysis of 30 randomly selected samples: 15 with detected MPU access, and 15 without. We replicated the steps in the pipeline manually, and verified the MPU access (or lack of access) through Ghidra's graphical interface, by checking for memory instructions targeting the MPU memory range (as in our analysis, there were no samples that used special MPU-related instructions). The results are in Table V in Appendix A. For all samples, our manual analysis found the same results as FIRMLINE.

D. Pipeline Performance

We now inspect all samples in our data set which did not time out at any stage in the pipeline (including code analysis samples). Out of the 21,755 samples we start our analysis with, none experienced a timeout during the preliminary analysis. However, 1183 time out during base address recognition, and 50 during base address detection and analysis passes. Accounting for all samples which either failed or timed out during an analysis step, FIRMLINE was able to process 3435 samples successfully.

Figure 3 shows the analysis time for this successfully processed samples. When considering the distribution of number of samples over analysis time, we see that the majority of samples that successfully completed analysis took less than 14 minutes to complete the pipeline. In general, analysis time was either very fast (2 minutes or less), or between 10 and 15 minutes, with some outliers. On the other hand, when considering the relative amount of samples for a given analysis time range, we see that the Linux and Tadiphone firmware had the largest variety in analysis time; firmware from other sources generally stayed in the 10-15 minute analysis time range. Firmware from the Monolithic Firmware Collection seems to be the fastest overall. Figure 4 shows a comparison of the analysis time of samples (horizontal axis) with their file size in kilobytes (vertical axis). There does not seem to be a strong trend of analysis time scaling with the size of a given sample, though the samples with the shortest analysis time were mostly small. There are outliers on both sides of the spectrum: larger samples that processed quickly, and smaller samples that took longer to process. This may indicate that even smaller samples can contain complex logic.

VII. Discussion

We obtain several insights from the results of our evaluation. First, we see that it is possible to analyze unknown firmware without relying on prior or domain-specific knowledge, although results lack behind approaches relying on such knowing. Additionally, the analysis time for our pipeline does not seem to correlate with the size of the firmware samples.

In our large-scale survey of non-Linux firmware, we learned that ARM is indeed one of the dominant architectures, at least for our dataset. The base address for non-Linux samples is most commonly non-zero. These insights can help focus future efforts and guide development efforts for analysis passes applicable to the most common architectures. During our security-related analyses, we observed that non-Linux ARM firmware generally does not seem to use special instructions to access the MPU; however, some samples do access the MPU via memory-mapped registers. Furthermore, some samples contain instructions to interact with an OS, possibly indicating that the target devices use an OS abstraction.

There are several threats to the validity of our results. The parts of FirmLine are dependent on each other: for example, when we detect the architecture of a sample, we assume this architecture for the sample throughout the rest of the analysis, and if the initial assumption is incorrect, our results at the end of the pipeline will be inaccurate. We also rely on the correctness of output from external tools, such as cpu_rec for architecture detection and Ghidra for disassembly. Similarly, we rely on Bincopy to correctly translate an IHEX file to a binary. If some of these intermediary tools produce incorrect output, this will negatively impact the validity of our results.

VIII. Limitations & Future Work

The work presented in this paper has three major limitations. The base address detection system is heuristic-based, and as such is not guaranteed to work for all firmware. Furthermore, base address detection and analysis passes are currently only applicable to ARM firmware, yet embedded devices, often use a different architecture. Finally, the analysis process requires a considerable amount of time, and the data was mostly collected manually. In this section, we will discuss these limitations in more detail.

In subsection IV-E, we explained our approach to detect the base address for firmware, which was modeled on that of FirmXRay [22]. An issue with this method is that it is heuristic-based, and relies on the convention of functions starting with prologues. We use instruction setting up the stack frame in function prologues to ‘signpost’ the start of a function. However, this may not always be the case: a branch target can essentially be any instruction. Hence, if a compiler (or a programmer) decides that no registers need to be saved to the stack before executing the rest of the function, there will be no such signpost, and hence fewer recognized functions. Therefore, determining a base address in this way is not guaranteed to work, which is why FirmXRay uses additional pointer types when resolving a base address. We tried to alleviate this by augmenting the list of found functions with functions identified by Ghidra’s built-in analysis passes. However, even with this optimization, our approach was only able to find a base address for a small fraction of FirmXRay’s samples, as described in subsection VI-A. It is also important to note that we use FirmXRay as ground truth for our experiments; However, obtaining actual ground truth for the analyzed firmware samples is challenging. Possible future work could investigate further ways to confirm the base addresses of samples, and augment the detection algorithm with that of Zhu et al. [50], or with recent advances in the field of function identification techniques (e.g., [51, 52, 53]).

Both base address detection and analysis passes are currently implemented using only instruction-level analysis targeting the ARM architecture. They search for ARM-specific instructions, and uses instruction alignment to simplify initial loading of the binary. Yet, as evidenced by the composition of our dataset described in section VI, a large portion of firmware does not necessarily target the ARM architecture. Hence, future work would extend FirmLine to generalize proposed passes or adapt them to different architectures. To support other architectures in analysis, one could explicitly modify the code analysis to search for the instruction equivalents on other architectures. However, a more generic solution would be to leverage binary lifting techniques to perform analysis on a shared intermediate representation.

For the most part, data collection in this study was manual. This limits the amount of data we can collect, so automatically searching for and downloading non-Linux-based firmware could be another direction for future work.

Finally, the analysis itself is rather time-intensive, as discussed in subsection VI-D. This is partially due to our reliance on Ghidra, which is a program primarily intended for interactive analysis and is not optimized for large-scale headless processing. Ghidra was also used by the state-of-the-art, FirmXRay, but it could be beneficial to port FirmLine to a tool targeted towards automated analysis, especially for large-scale analysis.

IX. Related Work

The most similar previous work is FirmXRay [22], which also inspired parts of our approach. The evaluation of FirmXRay includes a large-scale study of non-Linux-based firmware, specifically focused on vulnerabilities in Bluetooth Low-Energy technology at the link layer. To collect firmware, the authors downloaded mobile apps from Google Play, and extracted the firmware contained within, recognizing it through API functions and vendor-specific signatures. They narrow their scope to firmware from two specific vendors (Nordic and TI), use knowledge of the target’s architecture and the vendors’ SDKs to aid base address detection. The SDKs also allow them to resolve configuration values, and determine if those values lead to security issues. They analyze a dataset of 793 images, and find that 98.1% of devices have a static MAC address, 71.5% *Just Works* pairing, and 98.5% use insecure key exchanges. However, FirmXRay relies on vendor-specific knowledge in two major steps of their analysis: specific signatures that allow the authors to recognize bare-metal firmware when collecting data, and SDK functions and target device architecture when rebasing and disassembling firmware code. Hence, their techniques cannot readily generalize beyond firmware from Nordic and TI, especially if a firmware sample's vendor is not known or does not offer an SDK.

HEAPSTER [28] partially builds on the work of FIRMXRAY. The authors conduct a large-scale study of non-Linux-based firmware, though they mostly re-use the dataset from FIRMXRAY, augmenting it with samples with a known architecture retrieved from Fitbit. They apply the same technique as FIRMXRAY to load the firmware, but instead focus on issues in heap memory management libraries present in the firmware. They detect allocation and de-allocation functions by finding the sources of pointers passed to memory functions (e.g., `memcpy`), and checking the values returned when those sources are executed. Once locations of heap memory management functions are detected, they use HEAPHOPPER [54] to find vulnerabilities and generate a proof-of-vulnerability for those functions. In a dataset of 804 images, they found 11 heap management library families with 48 variations, and all contained at least one critical heap vulnerability. Since HEAPSTER is based on FIRMXRAY, the same limitations apply in terms of generalization.

Concurrent to our work, Tan et al. [55] conducted a large-scale study focusing solely on firmware targeting Arm Cortex-M systems. The data set comprises 1797 samples and the authors present similar findings in terms of MPU usage and presence of `svc` instructions. However, Tan et al. also assess additional security properties and note that multiple vendors leverage `svc` instructions to implement library calls, rather than operating systems with privilege separation.

Zhu et al. [50] develop an algorithm that uses the strings present in a given firmware sample to resolve a base address for the sample. They do not disassemble the binary, but rather search the byte stream directly for patterns representing ASCII characters, and for sequences encoding `LDR` instructions (which ARM firmware uses to load a string from a memory address). Their insight is that strings used in adjacent code will often be stored in a contiguous block of memory, and by correlating the addresses passed to `LDR` instructions with the offset of these string blocks, they can calculate a base address. Although this insight may not be true for all compilers or for hand-crafted assembly code, the authors were able to find the right base address for 9 out of 10 samples. The approach, as described by Zhu et al. [50], is specific to ARM firmware, but could be modified by searching for the byte encodings of similar instructions on other architectures.

Abbasi et al. [56] provide an quantitative analysis of exploit mitigation and security critical functions for 42 embedded operating systems and 78 core families. While their work focuses on deeply embedded systems (i.e., non-Linux based systems) and demonstrate the options available to firmware developers, little is known about the actual usage of the presented security features in the wild.

Furthermore, even while not their sole focus, some previous large-scale studies include a significant portion of non-Linux-based firmware. Costin et al. [7] conduct the first large-scale analysis of both Linux- and non-Linux-based firmware images (32,356 in total), employing a variety of techniques to obtain them (scraping vendor websites, custom search engines, and collecting user submissions). They try to crack password hashes contained in the firmware, and use a custom correlation engine to identify credentials and certificates shared among different images. Feng et al. [10] build the engine GENIUS to search firmware for functions, looking for code that is known to introduce a CVE, in a dataset of 8126 images. In BINARM [57], the authors build a database of vulnerable functions, and check if functions in 5756 firmware images are present in this database and KARONTE [58] analyzes multi-binary interactions for 899 samples. Finally, the authors of FIRMSEC [11] match code features of 32,817 firmware samples against code features in third-party components, and compare the components they find with a database of CVEs present in common third-party components. While their data set includes a considerable amount of non-Linux samples, the majority of these are not publicly available and the core analysis of FIRMSEC focuses on Linux-based firmware.

Linux-based firmware has been extensive subject to prior studies; hence, we only briefly summarize the large-scale static analyses approaches which are the most related to our work. FIRMUP [12] finds CVEs in firmware images by computing similarity between a query and target procedure, in a dataset of around 2000 executables. CRYPTOREX [59] use static backwards taint tracking to detect misuse of cryptographic functions, in a dataset of 1327 firmware images. Lastly, Yu et al. [60] measure the presence of vulnerability mitigations in 10,685 images, on both user- and kernel-level. These studies offer interesting approaches to analysis, but are heavily relying on abstractions provided by the Linux OS, and are hence not applicable to non-Linux-based firmware.

X. CONCLUSION

We presented FIRMLINE, an analysis pipeline for non-Linux firmware which aims to incorporate generic techniques to process unknown firmware. Since there is not much information about the ecosystem of non-Linux-based firmware, we conducted a large-scale analysis of such firmware. While our approach is not without limitations, we could load and analyze a more than 3000 samples in this manner. Our analysis indicates that while ARM is a prevalent architecture for firmware, a non-negligible amount of samples target other architectures. We further found that only a small fraction of ARM firmware seems to make use of memory protections or an operating system. We hope that our work forms the basis for further research: refinement of the presented techniques, and further investigation into the OS and security facilities of non-Linux firmware.

ACKNOWLEDGMENTS

This research was funded in whole or in part by UKRI EP/V000454/1. The results feed into DsbDtech. For the purpose of open access, the authors have applied a Creative Commons Attribution (CC BY) licence to the Author Accepted Manuscript of this work.

References

[1] L. S. Vailshery, "Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2023, with forecasts from 2022 to 2030."

[2] IoT Analytics, "State of IoT 2023."

[3] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou, "Understanding the Mirai Botnet," in *USENIX Security Symposium*, 2017.

[4] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas, "DDoS in the IoT: Mirai and Other Botnets," *Computer*, vol. 50, 2017.

[5] C. Lei, Z. Zhang, and C. Hu, "Old Wine in the New Bottle: Mirai Variant Targets Multiple IoT Devices," https://unit42.paloaltonetworks.com/mirai-variant-iz1h9/, 2023.

[6] R. Lakshmanan, "Active Mirai Botnet Variant Exploiting Zyxel Devices for DDoS Attacks," https://thehackernews.com/2023/06/active-mirai-botnet-variant-exploiting.html, 2023.

[7] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "Large-Scale analysis of the security of embedded firmwares," in *USENIX Security Symposium*, 2014.

[8] A. Costin, A. Zarras, and A. Francillon, "Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces," in *Asia Conference on Computer and Communications Security (AsiaCCS)*, 2016.

[9] D. Chen, M. Egele, M. Woo, and D. Brumley, "Towards Automated Dynamic Analysis for Linux-based Embedded Firmware," in *Network and Distributed System Security Symposium (NDSS)*, 2016.

[10] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[11] B. Zhao, S. Ji, J. Xu, Y. Tian, Q. Wei, Q. Wang, C. Lyu, X. Zhang, C. Lin, J. Wu, and R. Beyah, "A Large-Scale Empirical Analysis of the Vulnerabilities Introduced by Third-Party Components in IoT Firmware," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2022.

[12] Y. David, N. Partush, and E. Yahav, "FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[13] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang, and Y. Kim, "FirmAE: Towards Large-Scale Emulation of IoT Firmware for Dynamic Analysis," in *Annual Computer Security Applications Conference (ACSAC)*, 2020.

[14] Q. Li, X. Feng, R. Wang, Z. Li, and L. Sun, "Towards Fine-grained Fingerprinting of Firmware in Online Embedded Devices," in *IEEE Conference on Computer Communications (INFOCOM)*, 2018.

[15] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation," in *USENIX Security Symposium*, 2019.

[16] S. Thomas, F. Garcia, and T. Chothia, "Humidify: A tool for hidden functionality detection in firmware," in *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA '17)*, 2017.

[17] A. Fasano, T. Ballo, M. Muench, T. Leek, A. Bulekov, B. Dolan-Gavitt, M. Egele, A. Francillon, L. Lu, N. Gregory, D. Balzarotti, and W. Robertson, "SoK: Enabling Security Analyses of Embedded Systems via Rehosting," in *ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, 2021.

[18] C. Wright, W. A. Moeglein, S. Bagchi, M. Kulkarni, and A. A. Clements, "Challenges in firmware re-hosting, emulation, and analysis," *ACM Computing Surveys (CSUR)*.

[19] Arm Limited, "A32 Instruction Set Architecture," https://developer.arm.com/Architectures/A32%20Instruction%20Set%20Architecture.

[20] MIPS, "MIPS32 Architecture," https://www.mips.com/products/architectures/mips32-2/.

[21] Intel Corporation, *Hexadecimal Object File Format Specification*, 1988. [Online]. Available: https://archive.org/details/IntelHEXStandard/mode/2up

[22] H. Wen, Z. Lin, and Y. Zhang, "FirmXRay: Detecting Bluetooth Link Layer Vulnerabilities From Bare-Metal Firmware," *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.

[23] ——, "FirmXRay/dataset/," https://github.com/OSUSecLab/FirmXRay/tree/master/dataset, 2021.

[24] L. F. Group, "Repository of firmware blobs for use with the Linux kernel." https://git.kernel.org/pub/scm/linux/kernel/git/firmware/linux-firmware.git/tag/?h=20230210, 2023.

[25] erazor_de, "roccat Files - Roccat hardware support for Linux," https://sourceforge.net/projects/roccat/files/firmwares/, 2016.

[26] UCSB-seclab, "monolithic-firmware-collection," https://github.com/ucsb-seclab/monolithic-firmware-collection, 2020.

[27] Tadiphone, "Android Dumps," https://dumps.tadiphone.dev/dumps/.

[28] F. Gritti, F. Pagani, I. Grishchenko, L. Dresel, N. Redini, C. Kruegel, and G. Vigna, "HEAPSTER: Analyzing the Security of Dynamic Allocators for Monolithic Firmware Images," in *IEEE Symposium on Security & Privacy (S&P)*, 2022.

[29] N. Iooss and G. Campana, "Ghost in the Wireless, iwlwifi edition," in *SSTIC 2022*, 2022.

[30] Q. Hou, W. Diao, Y. Wang, X. Liu, S. Liu, L. Ying, S. Guo, Y. Li, M. Nie, and H. Duan, "Large-Scale Security Measurements on the Android Firmware Ecosystem," in *ACM International Conference on Software Engineering (ICSE)*, 2022.

[31] Software Freedom Conservancy, "Selenium WebDriver," https://www.selenium.dev/, 2010.

[32] Mozilla, "geckodriver," https://github.com/mozilla/geckodriver, 2014.

[33] M. Cecchetti, "SquashFS HOWTO," https://tldp.org/HOWTO/SquashFS-HOWTO/whatis.html, 2008.

[34] L. Granboulan, "cpu_rec.py, un outil statistique pour la reconnaissance d'architectures binaires exotiques," in *SSTIC*, 2017. [Online]. Available: https://github.com/airbus-seclab/cpu_rec/blob/master/doc/cpu_rec_sstic_english.md

[35] radare org, "radare2 source code," https://github.com/radareorg/radare2, 2023.

[36] XAMPPRocky, "Tokei," https://github.com/XAMPPRocky/tokei, 2015.
[37] Erik Moqvist, "bincopy," https://github.com/eerimoq/bincopy, 2015.
[38] Felix Domke, "bingrep," https://github.com/tmbinc/bgrep, 2010.
[39] ReFirm Labs, "Binwalk," https://github.com/ReFirmLabs/binwalk.git, 2013.
[40] D. Andriesse, A. Slowinska, and H. Bos, "Compiler-Agnostic Function Detection in Binaries," in *IEEE European Symposium on Security and Privacy (S&P)*.
[41] National Security Agency, "Ghidra Software Reverse Engineering Framework," https://github.com/NationalSecurityAgency/ghidra, 2019.
[42] J. Fitzpatrick, "An Interview with Steve Furber," *Commununications of the ACM*, 2011. [Online]. Available: https://doi.org/10.1145/1941487.1941501
[43] Arm Ltd., "The Arm ecosystem ships a record 6.7 billion Arm-based chips in a single quarter," https://www.arm.com/company/news/2021/02/arm-ecosystem-ships-record-6-billion-arm-based-chips-in-a-single-quarter, 2021.
[44] S. Murry, "ARM's Reach: 50 Billion Chip Milestone [VIDEO]," https://web.archive.org/web/20150916101815/https://www.broadcom.com/blog/chip-design/arms-reach-50-billion-chip-milestone-video/, 2014.
[45] AspenCore, "2019 Embedded Markets Study," https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf, 2019.
[46] National Security Agency, "Ghidra source code," https://github.com/NationalSecurityAgency/ghidra, 2023.
[47] Arm Limited, "Arm Armv8-A A32/T32 Instruction Set Architecture," https://developer.arm.com/documentation/ddi0597/2021-06/Base-Instructions/SVC--Supervisor-Call-?lang=en, 2021.
[48] ——, "Cortex-M3 Devices Generic User Guide," https://developer.arm.com/documentation/dui0552/a/cortex-m3-peripherals/optional-memory-protection-unit.
[49] ——, "Cortex-R4 and Cortex-R4F Technical Reference Manual r1p3," https://developer.arm.com/documentation/ddi0363/e/system-control-coprocessor/system-control-coprocessor-registers/c0--mpu-type-register?lang=en.
[50] R. Zhu, B. Zhang, J. Mao, Q. Zhang, and Y. an Tan, "A methodology for determining the image base of arm-based industrial control system firmware," *International Journal of Critical Infrastructure Protection*, 2017.
[51] C. Pang, R. Yu, D. Xu, E. Koskinen, G. Portokalidis, and J. Xu, "Towards optimal use of exception handling information for function detection," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2021.
[52] S. Yu, Y. Qu, X. Hu, and H. Yin, "DeepDi: Learning a Relational Graph Convolutional Network Model on Instructions for Fast and Accurate Disassembly," in *USENIX Security Symposium*, 2022.
[53] S. Kim, H. Kim, and S. K. Cha, "Funprobe: Probing functions from binary code through probabilistic analysis," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023.
[54] M. Eckert, A. Bianchi, R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "HeapHopper: Bringing bounded model checking to heap implementation security," in *USENIX Security Symposium*, 2018.
[55] X. Tan, Z. Ma, S. Pinto, L. Guan, N. Zhang, J. Xu, Z. Lin, H. Hu, and Z. Zhao, "Where's the "up"?! A Comprehensive (bottom-up) Study on the Security of Arm Cortex-M Systems," *arXiv preprint arXiv:2401.15289*, 2024.
[56] A. Abbasi, J. Wetzels, T. Holz, and S. Etalle, "Challenges in designing exploit mitigations for deeply embedded systems," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.
[57] P. Shirani, L. Collard, B. L. Agba, B. Lebel, M. Debbabi, L. Wang, and A. Hanna, "BINARM: Scalable and Efficient Detection of Vulnerabilities in Firmware Images of Intelligent Electronic Devices," in *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2018.
[58] N. Redini, A. Machiry, R. Wang, C. Spensky, A. Continella, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Karonte: Detecting Insecure Multi-binary Interactions in Embedded Firmware," in *IEEE Symposium on Security and Privacy (SP)*, 2020.
[59] L. Zhang, J. Chen, W. Diao, S. Guo, J. Weng, and K. Zhang, "CryptoREX: large-scale analysis of cryptographic misuse in IoT Devices," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2019.
[60] R. Yu, F. Del Nin, Y. Zhang, S. Huang, P. Kaliyar, S. Zakto, M. Conti, G. Portokalidis, and J. Xu, "Building Embedded Systems Like It's 1996," in *Network and Distributed System Security Symposium (NDSS)*, 2022.

Appendix

SHA-256 sum of file	Base address	MPU detected (manual)	MPU detected (pipeline)
f7752f6967487e7355e0aa1b015baaeeffaaa4794343e991f620fd3757f48879	114708	×	×
f3dea3328d8f5278bf5940132129a7effd59ddf01f623a45952b19df473af5fb	156452	×	×
508be2bf7bf48fd7c5191987374e91830dfb96febab4a9117204076cb8bb7c63	892546362	×	×
c7a3869642aec5d42d0b00543e07da588b8629e29314096f04e4699f760321af	78848	×	×
5a20c608351a6be29fa8c219510fbc00e1e44fb21bb0d11b47a3d7352b4bfc42	69756	×	×
e59f97b66f16ca46153c074a3f758d27b9bba333f880daf33d151c6c9828a132	98304	×	×
e530827b88d5ed6940d78dd1a84df7d0c15817bdfba79a92f67dd9cb243e254f	536872960	×	×
98fe82bbf94da0f0be4682cc07ba67e4a82236381ab62baf589b11a1f3605b34	180428	×	×
c9ffcc74c0d1586eadc4171a83e74740fa23fa744af2b5e034a8915011dd704c	120832	×	×
e115d302d3beac9c873c90d5f4eb6bfb65219e8ed8eae92d4bd2e9b51888eda8	167944	×	×
426a4c5e063a69be8e9cef0e895c9794e9645a6afd273a9887324a2e076bf425	672	✓	✓
c693bb3480a50d33d531f98dea92c33f81857a8471a3521e2e73181a7e50271b	36864	✓	✓
695f5f100e5065d15d71d0e9976dfe0f0644b6d0028fbced6f0f85b1f428cbe4	15360	✓	✓
cb4c5ba2ba08ab81ea03fa7851e3add1fbd8fd339378e3cbaa017ffcb746097c	15360	✓	✓
de90dca81bd8747de2cc35b4368b8951ddbdb1d3526a9963d6be49ea0d98bac7	83648	✓	✓
7674515f27ae059380cfc241467bd39050b580d2a4d16a4560ac41b8769d26f4	36864	✓	✓
dec4e83030976c267c79f4a6aaa25aef4a6374468b6788cd148b9e8ce1da7a63	13900	✓	✓
f3ae1502f1ab2e992d2b9161b92db61ca5dc38adcfdd9564248d34a2b58fdf10	36864	✓	✓
637158f8ef750ae7eab15a3de2ea27db3d10bf470d23bb0bd55b1897fe35cab2	15360	✓	✓
879256c0877b005743fff009ef4617657c6d9e518eb184a7c5b2ca5a20d3a5da	15360	✓	✓
d79b8fa0d9be90808e9e586c32894cb35c5b64ddf44055e51a48bc668ef8151a	90572	✓	✓
3195b393c89bd8a1f002b555d0c58067bd2e6c686ec2bb80954470531260d91a	15360	✓	✓
51758f2eb8cd0d38062ca6fe87f5e292ac16ae1bface1865e3a1d8916e905e4c	15360	✓	✓
cfc3c059b5a44e2671f01c4e185af7eac5b861bbec912f057e18ac2a37530d99	87312	✓	✓
4204cb40cb4456e379d0dcdfb46da69a15ab5f51c589f20089c64c7dda76fbc2	103952	×	×
7aed614ef1d02ac7e288ff968195f31d02323b2c695dab8ee0caf0bf53db12ac	1924	×	×
ad6fff54c1a68039ad238fc62efdcfc68efb78c87461a4aad7e247eb7baa3500	3328	×	×
fdea74bfa6195c90e831704e43fe12473c9395e69a9d7f44d5e91e744a7aae28	536881152	×	×
302c325488d204297d699b7bb179ee0d42f5fc82c7f1a68b6ef010612fe3b56a	139264	×	×
97036720fe2ee8940f561b2c12de323b2ed4b1c779caee2f605c65a5a1f30b82	134245376	✓	✓

Table V: Comparison of MPU detection results for the pipeline with a manual analysis (30 random samples).

SHA-256 sum of file	**Base address**	**OS detected (manual)**	**OS detected (pipeline)**
a4b6857b4b4050cdacc101bfa2fa86b1142a257d95a6b0a206c044e8c637f8c3	122880	✓	✓
a2254573688b531a0d477cc2d7a2e01bbe2fd0a9cd8c4c760999f594d3aa6172	197632	✓	✓
95b95f857a5e70167ccf28e84dc78e816b221b91bcf5e9f2150d5969cca5e695	314628	×	×
d4a71f6035be48344abff8c2da813a3d4b6f82c880bb4b63b58f1b38a8ca3e37	0	✓	✓
f05a049d14a0a429ed40df6134ef7badc41da9cf579902c1d1debfbafab6340b	0	×	×
0c95c5616861856b055209a05270c0cc75cb500daa550a659b275e124ea334aa	290	×	×
d711fb2b08057fede77800a1ef733ce94baf89759caa1361bb441e6682e42f1d	138240	✓	✓
a16e54b2548e7883749d851cfbf8d8a245373ffab064c6ee9382f0387724318d	77824	×	×
337d1730aab2ab7afba5f349dccb5250e36994bdd298dafd887b1e3f5282c457	757030174	×	×
6436f7f4d16cb2f8ccc1a0cb2015928944f37648831ee7963291dcc714e376bf	429572	Maybe (“Tsk_Ssvc”, “Elf_Ssvc”)	×
94bf44c56fdd4345ed22d99f13510d4ddf7276fadf12fd717622d6110f3cbb68	536662008	✓	✓
5d3e29babfa44b1ca56ca98abb32361b6781dbce7a0daaea05ab249d5639ad0e	133636	✓	✓
dff70fbf0cf7e00ad6d8c31c823b59264166dad301647846abaf32c8030144bf	114824	✓	✓
c6fc9b645d416eafbd4d1e91089dc269a8031a91cbbd16b67b34384c67a89418	114824	✓	✓
29be79731ebf2eb193f0159ae887e8af049e4ccd9a2e3298358a1ef982c3ccd2	0	×	×
7a2dca4888fbe66722cfa1d0c584c1f9d3d1d751097260207ff04ea18d3ac1fc	132098	✓	✓
98fe82bbf94da0f0be4682cc07ba67e4a82236381ab62baf589b11a1f3605b34	180428	✓	✓
2d4a18a23221919f27ba8d32ed3c01aba78eeaf93b743fe5184660bf11fbfab7	0	×	×
de90dca81bd8747de2cc35b4368b8951ddbdb1d3526a9963d6be49ea0d98bac7	83648	Maybe (“Unknown SVC 0x%02lX called at 0x%08lX”)	×
aed0923f52518890e9f4a64bb1cfeea0dde1238e33be3500dad13e6cc469f892	15360	Maybe (“Unknown SVC 0x%02lX called at 0x%08lX”)	×
2617f34ee6a43a11107a4627bbeb69cae8fd4a66fce11b79acaaeb961ca210c6	0	×	×
340f3f39173a0d2239a99645345e60abbeb45791a4279d625878c696f72c7d43	88564	×	×
b1c6f20bfd5ab8b3d775ee62d10f9dfb8febc758e7cbef3bd1f0ecd54bac8809	29696	×	×
ac40481cd4317bf5a4f0352a37a4d3237fe9782e156c9a05d2984e69ac15bf4c	5423186	×	×
5b1a3037dd73d2c77ef7c4c9fda938ed4e48e3f6a4e99186ab7858503a918b05	536803326	✓	✓
6304c428233d18399a9e05d24dbcebe48948a69df39cc714a23d86f31cb1a16b	0	✓	✓
5fae530df7f25b843c3c4a3f456a4a4932849f0f38f1dc16c730ae7d1d120d5e	1924	✓	✓
867a5c0a23a11fc25d7623fe1e8c6dad0e9a43ca7839695bb7923665778335b3	1216	✓	✓
a60be4e9fa8e2917f4742fe85fd48627a4424a1c471aaa698d48744dbd58ff5d	0	✓	✓
361e4f2ac4b7149fd0259931c252b6c40a1605c570d666598348f769aa85b7a2	5453906	×	×

Table VI: Comparison of OS detection results for the pipeline with a manual analysis (30 random samples).