Literature Study

A Classification-Based Survey of Existing Firmware Research

by

Alexander Balgavý
(Student Number: 2619644)

June 19, 2023

Certified by ....
Marius Muench
Dr.
First Supervisor

Certified by ....
Enrico Barberis
M.Sc.
Daily Supervisor

Certified by ....
Cristiano Giuffrida
Associate Professor
Second reader

A Classification-Based Survey of Existing Firmware Research

Alexander Balgavý
Vrije Universiteit Amsterdam & Universiteit van Amsterdam
Amsterdam, NL
a.balgavy@student.vu.nl

ABSTRACT

Embedded devices are a common, sometimes invisible, part of our lives. Like any other device, they can have security vulnerabilities. The implications of these vulnerabilities may be more serious and immediate than on desktop or mobile systems, as embedded devices are frequently used in critical applications, such as the medical and industrial sectors. In this paper, we conduct a systematic study of past analyses of embedded device firmware. We find that most analyses are small-scale in terms of the number of firmware samples, and those that are on a larger scale tend to focus on Linux-based firmware. Still, a large number of embedded devices use non-Linux-based firmware, so this group should not be neglected. Our results indicate that a large-scale study of non-Linux-based firmware, along with the development of generic analysis techniques for unknown firmware, could be a promising direction for future research.

1 INTRODUCTION

Embedded systems are continuously becoming a larger part of our lives. Most commonly, they are seen in commercial off-the-shelf (*COTS*) networked devices that are part of the 'Internet of Things' (*IoT*), but they may also be used in industrial control devices, medical devices, and other sectors. While it is not possible to calculate the exact number of currently deployed devices, some estimate that by 2020 it would have exceeded 20 billion [29]. Such widespread use of embedded devices makes their security all the more important, especially if used in life-critical applications.

Just like desktop and mobile systems, embedded devices can have security vulnerabilities, either due to hardware, or due to issues in the software running on top of it. For example, the ZuoRAT [2] malware targets routers at small and home offices, and enables installation of other malware on connected hosts via DNS and HTTP hijacking, as well as allowing the attacker to conduct in-depth reconnaissance of the network. Project TEMPA [20] shows a method that exploits the Near-Field Communication (*NFC*) and Bluetooth Low Energy (*BLE*) based unlocking system of Tesla cars to clone the owner's key and steal the car, as long as they are within range. The Mirai [1, 24] botnet, mainly containing vulnerable IoT devices with a peak number of 600k infections, launched a distributed denial-of-service attack (DDoS) and caused multiple-hour outages at large companies, notably Twitter, Netflix, Reddit, and GitHub. Considering the prominence of vulnerabilities in embedded devices and their potentially devastating effects, it is necessary to study the security of these devices and their associated software, or *'firmware'*.

There are multiple definitions for the term 'firmware', but IEEE [22] defines it as "the combination of a hardware device and computer instructions and data that reside as read-only software on that device." In general, 'firmware' is used to describe the software that is stored on, and controls, an embedded device.

Past research into embedded device firmware has focused mostly on specific device types (e.g., routers). In terms of large-scale analyses, most have been carried out on firmware that is based on Linux, which is only a subset of all possible device firmware: some may have a custom kernel, or no kernel-user separation at all. To the best of our knowledge, there has not been a comprehensive study of existing firmware research focusing on firmware that is not based on Linux.

Hence, our main contributions in this paper are:

- a labeled and organized corpus of past firmware research, along with analysis scripts,[1]
- a summary and analysis of past studies of firmware at both large and small scale.

2 BACKGROUND

This section explains concepts and techniques relevant to the discussion of firmware analysis studies.

2.1 Unpacking, disassembly, and decompilation

Firmware images generally consist of one or more executable binary files, and potentially other types of files (images, PDF manuals, text, keys/certificates, etc.). These files are packaged, or *'packed'*, into a single file for distribution. That single file can be in a disk image, or a standard archive format, potentially compressed and/or encrypted, such as a `tar` file compressed with `gzip` or a zip file. It can also be a binary object, with a standard header or a custom header format. Furthermore, the binary executable itself can be packed or obfuscated. In standard use, a packed binary executable will generally unpack itself using a code 'stub', but in analysis, this is a separate preliminary step. *Unpacking* refers to the extraction of these inner files, and the unraveling of binary code to a form amenable to analysis.

After unpacking, a binary executable consists of byte streams: some parts may be data, and others may be platform- and architecture-specific operation codes with arguments that instruct the processor to carry out operations. The first step in analyzing the executable is identifying instructions: if

[1] https://github.com/thezeroalpha/msc-literature-study/blob/master/analysis/analysis.org

the firmware uses a standard header that stores information about the location of the code, we can easily determine this, but if the header format is custom, we need to first determine where to start analysis. A different base address yields a very different disassembly, so this step is key for code analysis. Once we know the location of the code, we can proceed to *disassembly*: converting the byte streams to assembly instructions.

It is also possible to take this further with more abstraction. One option is *decompilation*: converting assembly instructions (which are still platform- and architecture-specific) to a higher-level programming language, such as code in a pseudo-C syntax. Another option, more commonly used in analysis of firmware (either directly or as an intermediate step before decompilation), is lifting the assembly code to a slightly higher-level intermediate representation (*IR*). An IR, such as VEX and LLVM, is abstract enough to eliminate architectural differences and allow unified reasoning about program behavior, but not as abstract as programming languages intended for users.

2.2 Types of analysis: static, dynamic, and symbolic execution

Program analysis comes in different forms: static, dynamic, and hybrid. Static analysis examines the disassembled code without running it. A common method of static analysis is building control flow graphs, or *CFG*s: graphs where nodes represent basic blocks, and edges indicate the possible control flow between basic blocks. A *basic block* is a series of instructions with one entry point and one exit point. Another option for static analysis is building a *call graph*, which is similar to a CFG, except it only shows function calls. Data flow analysis is also common: tracking which data definitions can reach a certain point in the program. A metric that's often used to determine the complexity of a function statically is *cyclomatic complexity*: the number of linearly independent paths within the function, computed from its CFG. For example, a function with a single-condition if statement would have a cyclomatic complexity of 2, because there are two linearly independent paths: one where the condition is true, and one where it is false. Static analysis is often combined with *taint tracking*: a method to determine which tainted data (commonly representing attacker-controlled data) can reach various points in the program, particularly 'interesting' points such as privileged code. However, static analysis may not be enough to properly analyze a program, especially if the program is obfuscated/encrypted in some way, or if an execution path is computed at runtime and cannot be discovered by inspecting the program in isolation. Furthermore, not all execution paths visible in static analysis may actually be reached or reachable in execution.

Dynamic analysis, on the other hand, focuses on the execution of the target program, i.e., how it actually behaves at runtime. On desktop systems, this is often done via instrumentation: modifying the program and injecting code to collect runtime information. However, for embedded devices, instrumentation is often not feasible, because of differences in architectures and sometimes storage limitations (instrumentation increases the size of a program, and there may not be enough space on the device to store an instrumented version of the program). Therefore, a common technique for dynamic firmware analysis is executing firmware in a virtual environment. One option is to use an emulator; however, embedded devices can interact with a wide variety of peripherals, which may not be easily implemented in an emulator. Another option is rehosting, which is software-centric: hardware is either modeled in software, or the hardware interactions are forwarded to the actual peripherals. As in static analysis, dynamic analysis can also be combined with taint tracking, to see which points of the program tainted data reaches during actual execution. Another common dynamic analysis technique is fuzzing: sending various (e.g., malformed) inputs to the firmware, with the intent to crash it or to trigger a vulnerability.

Finally, symbolic execution is a hybrid between static and dynamic analysis. It does not require the actual device for execution, but rather 'pretends' to execute the code by modeling device state (registers, memory, etc.) in software, and acting out operations that executed instructions would normally do on the device. Instead of values, a symbolic execution system uses symbols, which are bound by constraints – mathematical restrictions on the possible values for the symbol. It then uses an SMT solver to solve the constraints and obtain values satisfying them. The advantage of symbolic execution is that it does not need the actual device or a sufficiently emulated version of the device to 'execute' the code; in fact, a symbolic execution engine could itself be considered an emulator or virtual execution engine. However, symbolic execution can be limited when dealing with I/O, and the number of possible paths to be explored increases exponentially (a problem known as 'path explosion'), so symbolic execution of an entire program can become intractable.

2.3 Interaction of firmware with hardware

Embedded devices frequently interact with peripherals: hardware input and output devices to communicate with the 'outside world'. These can be off-chip (separated physically from the processor and connected via a bus), or on-chip (sharing the processor's chip and directly interfaced). The processor can communicate with peripherals in several ways. One option is direct memory access (*DMA*), which allows transferring data directly from CPU memory to a peripheral. The most frequent interaction with peripherals occurs via memory-mapped I/O (*MMIO*), which maps registers of the peripheral device into a block of the processor's memory. Another option is port-mapped I/O (*PMIO*), where each I/O device is assigned to a port, and the CPU uses special instructions to copy data between its registers and a given port. Data can be read from a peripheral either based on interrupts (i.e., when data arrives, raise interrupt and read it), or based on polling (continually check the status register, and if data is available, read it). MMIO and PMIO are processor-initiated; a device may also itself initiate communication with the CPU, using a hardware interrupt. Hardware interrupts are unidirectional, and only communicate to the CPU that an event occurred and should be handled.

3 METHODOLOGY

To make sense of the research landscape in the area of firmware analysis and identify potentially interesting areas for further research, we must be able to make quantitative observations about existing research. To achieve this, we opt for a classification-based approach, where we categorize previous research across several variables.

3.1 Previous classification efforts

There are several ways to classify firmware. Muench et al. [27] provide one option: they separate embedded systems by the type of operating system that they use. This yields three classes: Type-I (general purpose operating systems), Type-II (custom operating systems designed specifically for embedded devices), and Type-III (no operating system abstraction). Using this framework, we could categorize research based on the types of embedded systems for which the analyzed firmware is intended. However, we often found it difficult to distinguish which of the embedded system types were being discussed purely based on information in the research paper, particularly when trying to determine whether a system was Type-II or Type-III.

As another option, Costin et al. [7] separate firmware by operating system (e.g., Linux, VxWorks, Windows CE). However, because a large amount of research involving firmware that does not use Linux includes samples without an operating system, this classification would not capture information for a significant amount of our data.

3.2 Our categorization

First, we separate studies by size: those considered large-scale, and those considered small-scale. There is no established definition of a "large-scale" study; in this paper, for simplicity, we consider a large-scale study one that treats more than 500 samples. This amount of samples is sufficient to guarantee for variety in the dataset (as smaller datasets may be biased towards specific vendors/devices), and multiple studies have shown that their analysis can scale up to and beyond this point.

The classification approaches discussed in Section 3.1 were developed to classify firmware. Since our goal is not to classify firmware itself, but instead to classify research works dealing with firmware, those classification systems do not fit our purpose. Hence, we developed our own classification, across several variables: firmware types, analysis types, scraping approaches, device types, instruction set architecture, whether and where the dataset is available, and what parts of the firmware were analyzed. We can split these variables into categorical data and numerical data. Categorical data, except for dataset availability, are multi-valued variables; i.e., the values for each variable are not mutually exclusive. Numerical data captures information about the number of samples that were initially retrieved, and how many were actually used in analysis (after any preliminary filtering).

3.2.1 Linux-based versus Non-Linux-based firmware. We split the firmware (and hence research of firmware) into two types: Linux-based (LB), and non-Linux-based (NLB). LB firmware is designed to execute on top of an operating system that is based on the Linux kernel [38]. NLB firmware either executes on top of a different operating system (e.g., VxWorks), or has no OS abstraction whatsoever. A study can be labeled as both LB and NLB if it analyzes both types of firmware.

3.2.2 Analysis types. The 'analysis types' category captures the style of analysis that the research in question conducted. It indicates whether the authors performed static analysis, dynamic analysis, symbolic execution, or a combination of the three. It also shows whether the study included taint tracking.

3.2.3 Analysis focus. We also tracked the parts of firmware that the papers analyzed. Some research analyzed the actual code of the firmware; some only the configuration or credentials stored in the firmware. Other papers analyzed the interfaces of the firmware – that is, they interacted with the methods that the firmware uses to communicate with the 'outside world' (e.g., web interfaces, ports, protocols).

3.2.4 Obtaining firmware. The 'scraping approaches' category contains the methods that the authors of a particular source used to retrieve firmware samples. This could mean downloading them from vendors' websites or FTP servers, using an app-based approach, using a custom search engine (such as the one offered by Google,[2] extracting them directly from the devices, compiling them from source code, or allowing users to submit firmware for analysis. Some studies reused existing datasets from previous research, and some did not specify the methods they used.

3.2.5 Device types: sectors. We opted to separate the analyzed devices by sector, i.e., in which area those devices are commonly used. The 'personal' sector contains devices that a single person would use and would carry with them (e.g., wearable, phone, tablet, smart eyeglasses). The 'IoT' sector includes internet-connected devices that a person would place in their home or on their belongings (e.g., wireless tag, printer, switch, routers and access points, network-attached storage). The 'computer peripherals' sector contains components of computers, or devices intended to be used with a computer (e.g., device drivers, bootloaders, batteries, storage devices, network interface cards, mice, and keyboards). The 'industrial' sector includes devices used for manufacturing or control in industry (e.g., agricultural equipment, reflow oven, heat press, industrial control system devices such as programmable logic controllers). Finally, there is a 'medical' sector for devices used in medical contexts (e.g., a blood pressure monitor), and an 'other' sector for devices that do not fit

[2] https://programmablesearchengine.google.com/about/

in the other categories (e.g., drone, card reader, development boards, gateway device, single board computer).

3.2.6 CPU information. We also keep track of the instruction set architectures (*ISA*s) of the CPUs for which the analyzed firmware is intended. The only coalescing we did in this category was grouping versions of ARM (e.g., ARMv7-Thumb, ARMv8A) under the same 'ARM' label. If a paper did not explicitly state the architecture, we researched the specific processors of the devices that the authors used, and noted their architectures.

3.2.7 Reproducibility: availability of data. We verified whether the dataset that the authors used is (still) available. The data can be made available in several states. It is ideal when the authors of a study directly provide their dataset, because then the version of data is guaranteed to be the same as when the authors used it. However, this is not always the case in practice. If the authors used a dataset compiled in previous research, they may only link to that dataset. It could also be that the authors only release a part of the dataset. Unfortunately, in some cases, the firmware samples they used may not be available at all.

4 OVERVIEW OF LITERATURE

In this section, we give a summary of the literature relevant to this study.

4.1 Large-scale studies of NLB firmware

In total, we look at 6 large-scale studies, separated by whether they only analyze NLB firmware, or both LB and NLB firmware. All large-scale NLB firmware studies used a static approach, with only HEAPSTER [16] also incorporating symbolic execution.

4.1.1 Both LB and NLB firmware. One way to analyze firmware is to look at sharing of code and/or data. Costin et al. [7] investigate shared data among firmware, particularly credentials and certificates. They scrape the first public large-scale analysis of both LB and NLB firmware images, retrieving data from vendor websites, FTP servers, and mirror sites using a custom crawler. They also collected data via custom search engines and user submissions of firmware. The dataset contains 32,356 firmware images spanning the personal, IoT, and industrial sectors. Then, the authors look for password hashes and attempt to crack them via dictionary and brute-force attacks. They also develop a correlation engine to find similarities between firmware images, such as common credentials and certificates, helping to identify shared security issues across different images. The authors discover 38 previously unknown vulnerabilities in 693 of the images; by correlating similar files stored in the images, they extend these findings to over 123 different products.

Another option is to search the code of unknown firmware for instances of known vulnerable patterns. In *Scalable graph-based bug search* [14], the authors search for functions similar to query functions known to introduce a CVE. Their search engine, Genius, then uses these vectors to find functions similar to a query function. They encode CFGs as numeric feature vectors, based on properties of functions. The authors observe that this engine can finish a search across 8126 firmware images, scraped from vendor websites, within an average of one second. Furthermore, only in the top 50 candidates, it found 38 potentially vulnerable firmware images; the authors confirmed 23 of them by manual analysis.

Instead of searching for individual vulnerable elements, one could build a database of such elements, and then check if parts of unknown firmware appear in that database. The authors of BINARM [34] build a database of vulnerable functions for firmware targeting intelligent electronic devices: because vendors are obligated to publish licenses of open source software that they use, it's possible to determine which open source projects are used in the firmware, and those projects can be correlated with the CVE database to find vulnerabilities in the firmware. Then, they create an engine to check if a given query function is in this database, and hence check if it's vulnerable. Their results show that the average accuracy in identifying vulnerable functions is 0.92, and their detection engine can speed up existing approaches by 3 orders of magnitude. In their evaluation on 5756 firmware images (obtained with a website scraper from vendor websites and FTP servers), their prototype detects 93 real CVE vulnerabilities.

This is also possible at a higher level; i.e., instead of looking for vulnerable functions, one can consider the third-party components (*TPC*s) as a whole. In FirmSec [45], the authors create a database containing versions of TPCs (such as BusyBox or OpenSSL) and their related CVEs. They collect images with a web crawler that retrieves data from vendor websites, FTP websites, the community (forums and GitHub repositories), and a private firmware repository of a smart home company ("TSmart"). In analysis, their system matches code features of firmware against those extracted from TPCs to identify which TPCs are present in the firmware, and then queries the database to find whether any of them introduces a CVE. If any of those TPCs are known to have vulnerabilities, the firmware itself is also vulnerable. They use this system to conduct the first large-scale analysis of the usage of TPCs in firmware, and detect 584 of them in a dataset of 32,817 images, identifying 128,757 vulnerabilities in total.

4.1.2 Only NLB firmware. As part of the first large-scale study focusing *solely* on NLB firmware, FirmXRay [40] looks at Bluetooth Low-Energy vulnerabilities caused by configuration at the link layer. To load the firmware, they resolve its base address by relating memory pointers to their targets, and choosing the offset with the most pointers with valid targets. They use knowledge from the vendors' software development kits (*SDK*s) to identify some of these pointers, and then to resolve configuration values. It is then possible to determine if the obtained configuration values lead to vulnerabilities, such as identity tracking due to a static MAC address, or active MITM vulnerability due to *Just Works* pairing. In a dataset of 793 images extracted from mobile apps downloaded from Google Play, they found that 98.1% of devices have a random static MAC address, 71.5% *Just Works* pairing, and 98.5% use insecure key exchanges.

Issues in firmware images may not only surface in configuration, but also in the lower-level software components, such as heap memory management libraries. HEAPSTER [16] automatically identifies the heap memory allocation library used by a monolithic firmware image, and tests its security with symbolic execution and bounded model checking. To properly load the image into an emulator, they use the same base address extraction technique as FirmXRay [40]. They identify allocation and de-allocation functions by locating the sources of pointers passed to functions such as `memcpy`, and checking the values returned when those sources are executed. Using HeapHopper [11], they generate a proof-of-vulnerability for any vulnerable allocator or de-allocator. In a large-scale evaluation on real-world samples, the authors reused an existing dataset from FirmXRay [40], adding 5 firmware samples downloaded from Fitbit. In this dataset of 804 images, they identified eleven heap management library families with 48 variations, all of them vulnerable to at least one critical heap vulnerability.

4.2 Small-scale studies of NLB firmware

We also include small-scale NLB firmware research in our corpus.

4.2.1 Both LB and NLB firmware.

Small-scale studies of NLB firmware are split approximately equally between static and dynamic analyses.

Static approaches. Code search analyses have also been used at a smaller scale; for example, one way to efficiently recognize code is by creating a 'signature' based on features of the code. In *Cross-Architecture Bug Search* [31], the authors calculate signatures for known bugs, which are derived from input and output variables of basic blocks. They then search for these signatures in a dataset of 60 binaries extracted from vendor-compiled router firmware, targeting MIPS and ARM. Due to their use of an IR, their prototype can find Heartbleed bugs [37] regardless of the underlying instructions set, and also found backdoors in closed-source MIPS and ARM firmware.

Similarly, discovRE [12] also searches for bugs, but with a somewhat different approach: starting with a vulnerable binary function, the system identifies similar functions in other binaries. To determine similarity, they use properties of functions as features, and pass these features through two filters to narrow down candidates. They evaluate their prototype on an existing dataset compiled by Pewny et al. [31] and two additional firmware images (in total, 62 samples). Their prototype could identify Heartbleed [37] and POODLE [26] vulnerabilities much faster than the state-of-the-art approaches at the time.

Some vulnerabilities are triggered by user input, so Karonte [33] analyzes data flow in firmware to find insecure flows of attacker data. The system applies various heuristics to find 'border binaries' (binaries that interact with the outside world), builds a graph of data flow between binaries in the image, and then uses static taint analysis to track data propagation in the graph. If Karonte finds an insecure flow of attacker-controlled data (such as possible memory corruption or attacker-controlled loops), it reports the issue for inspection. On a dataset of 53 firmware samples scraped from official vendor websites, Karonte produced 87 alerts, out of which 51 were true positives (including 46 new zero-day bugs).

Dynamic approaches & symbolic execution. Firmalice [35] also analyzes data flow, but in a more hybrid way than Karonte [33], using symbolic execution to determine if program can reach a pre-defined privileged point. To find the base address, Firmalice uses jump tables: lists of absolute code addresses stored in the device's memory. In parts of the program ranging between a privileged point and an entry point, Firmalice tries to identify user inputs which reach the privileged program point, using symbolic execution and constraint solving to find such an input. The authors were able to detect authentication bypass backdoors in two out of three commercial device firmware.

A challenge for the analysis of NLB firmware is a lack of feedback: even if there is a problem, embedded devices may not make these effects as visible, due to their more streamlined design. In fact, Muench et al. [27] show in their paper that memory corruptions on embedded devices often have effects that are less visible compared to desktop systems, reducing the effectiveness of dynamic testing techniques, particularly fuzzing. Since their aim is not to find vulnerabilities but to analyze their effects on embedded systems, they themselves introduce several memory corruption vulnerabilities (but not handling of those vulnerabilities) into the code. They look at a total of three devices, one for each of the device classes they define (Type-I, Type-II, and Type-III), as discussed in Section 3.1. They find that the fewer features an embedded system has, the less likely it is to detect memory corruptions, with "Type-III systems" (those without an OS abstraction) either hanging or not showing a visible effect.

4.2.2 Only NLB firmware.

There has also been focus on small-scale analysis only of NLB firmware; some of this research is scoped to only one or two specific devices, such as a keyboard or mouse. In this section, we summarize only small-scale NLB research that analyzes a larger number of samples (though not enough to be considered large-scale), because such research is closest in scope to our literature study.

Static approaches. Some focus has been on improving the techniques prerequisite for, or related to, firmware analysis; this is particularly important for NLB firmware, considering the lack of standards. Before analysis, the firmware must be disassembled starting from its base address, and Zhu et al. [47] describe a method to automatically find the image base of firmware, with a focus on industrial control systems. They present three static algorithms: FIND-String to obtain offsets of strings in firmware, FIND-LDR to find string addresses loaded by `LDR` instructions, and DBMSSL, which uses outputs from the previous two algorithms to find the base of a firmware image. They do not disassemble a given binary; instead, they search the bytes directly for ASCII strings and sequences encoding `LDR` instructions. They then use the discovered strings and addresses used as operands to `LDR` instructions to calculate a base address. Applying this approach to ten firmware samples collected from vendors, the correct image base was found in nine of them (for the last image, the approach was not applicable, because it did not contain enough strings, or it was encrypted/compressed, or it used the `ADR` instruction instead of `LDR`).

Dynamic approaches & symbolic execution. Another area where improvement of techniques is important, especially for dynamic analysis, is handling of hardware. Instead of incorporating physical hardware in analysis, the hardware can be modeled in software. Pretender [18] uses observations of interactions between firmware and hardware to automatically model the peripherals, allowing executing firmware in a fully emulated environment. They evaluate the prototype on a dataset of six firmware, retrieved from vendors and created manually, and with this evaluation, they showed that Pretender can achieve good execution coverage in a virtual environment that is automatically generated.

Instead of fully emulating or rehosting firmware, one may conduct semi-dynamic analysis with symbolic execution. However, a complete analysis of code may sometimes be intractable, as the amount of paths to be explored increases exponentially. FIE [10] addresses this problem with state pruning (removing redundant states) and memory smudging (if any memory locations in a new successor state have been modified more often than a provided threshold, the memory location gets a wildcard value, collapsing all future possible states of this variable into one). Also, existing symbolic execution engines such as KLEE do not always support the variety of architectures targeted by firmware, so the authors of FIE develop a modular way to add support for these architectures, specifying e.g., the memory layout of the target architecture (in their case, MSP430). When running, FIE either visits every possible state in symbolic execution, hits the chosen time limit, or finds a security/safety violation; in the latter case, it outputs a detailed description of the violation. Applying FIE to a corpus of 99 firmware programs scraped from vendor sites and GitHub, the tool verifies memory safety for the majority of them and finds 21 bugs.

If one decides to fully execute the firmware in a virtual environment, there is the question of what should be done about hardware interaction. Avatar [43] keeps the hardware in the loop, forwarding memory accesses to the real devices where needed, and the authors use it to analyze firmware with the goal of detecting compromise of execution by an attacker. The prototype can communicate with a device using different protocols depending on the device's debugging support. If the hardware interaction is only limited, Avatar allows recording those input/output operations and replaying them in the next execution, without the need to actually use the hardware. Avatar can perform a wide range of analyses, but in their evaluation, the authors focus on finding input commands that could be used to reach arbitrary execution or control flow corruption. They test the system on three samples obtained from vendors, and were able to detect an arbitrary execution vulnerability in one of them.

Redini et al. [32] also investigate the compromise of security by a potential attacker, but in lower-level components: bootloaders. BootStomp locates areas of modern device bootloaders where attacker input can compromise the bootloader's execution or security features. They use taint analysis through symbolic execution, and look at exploitation through attacker-controlled storage, by triggering a memory-corruption vulnerability or by unlocking the bootloader. With this approach, they find six previously unknown vulnerabilities in five bootloaders of commercial devices.

Another area of investigation is the use of protocols in firmware [15, 21]. Both of these studies use symbolic execution and allow queries about protocols, but with a different scope: FirmUSB [21] focuses on the USB protocol, while ProXray [15] generalizes to any (unknown) protocol. FirmUSB uses domain knowledge about the USB protocol to guide symbolic execution, and allows two semantic queries about its use in firmware: 'claimed identity' (whether claimed device characteristics reflect the real characteristics) and 'consistent behavior' (whether the claimed functionality is actually carried out by the device). This lets the system detect attacks like BadUSB [28], which lead to inconsistent behavior. On the other hand, ProXray [15] is able to learn a protocol model from known firmware, and apply it to unknown firmware to recognize protocol-relevant fields and detect functionality. This enables queries in the form of protocol-related constraints (such as specific code numbers, packet types, length, etc.), and these queries guide symbolic execution of the firmware to obtain an answer to the query. ProXray's dataset consisted of 29 images scraped from vendors, with 23 used as the training set. They applied the tool to the USB and Bluetooth protocols, and were able to identify USB functionality automatically within all six unknown USB firmware in the testing set, with a significant speedup.

A problem with dynamic analysis is that often knowledge about a device's design or the design of the underlying hardware is a prerequisite, and is not standardized for NLB firmware. P^{2}IM [13] alleviates this issue, by automatically generating 'approximate emulators'. Since firmware can execute on an emulator without real/fully emulated peripherals as long as it receives acceptable input from peripherals when needed (i.e., input that does not crash the firmware), the authors instead automatically generate models with the minimum required functionality. For evaluation, they used a dataset of 80 firmware, retrieved from vendors. Using the AFL fuzzer as the input source and a memory error detector, their results unveil 7 unique bugs; however, this approach does not model direct memory access which may occur with peripherals, particularly for embedded devices.

In some cases, a device might use a Hardware Abstraction Layer (*HAL*), which the analyst can incorporate to simplify modeling of hardware in software. HALs are software libraries that provide access to high-level operations on the device, hiding details of the particular chip or system. HALucinator [6] uses these HALs, which are already provided by firmware vendors, as a basis for rehosting and analysis of firmware. The authors provide generic implementations of these functions as high-level replacements for HAL functions. They use this system for interactive emulation and fuzzing, and successfully applied it to crash firmware and uncover bugs in its code. For this approach to be successful, the firmware has to use a HAL, which must be available to the analyst, and separate handlers and peripheral models are required for each HAL.

4.3 Only Linux-based (LB) firmware studies

As discussed in Section 3.2, we separate studies by size, and treat large-scale and small-scale studies separately in this section.

4.3.1 Large-scale studies of LB firmware. From the LB studies we analyzed, 8 of them have been large-scale [3, 8, 9, 23, 25, 36, 42, 44].

Static analysis. When searching and identifying binaries, performance is particularly important at a large scale. Since in embedded devices, firmware information (particularly the vendor and version) is associated with security vulnerabilities, Li et al. [25] propose a method of fine-grained fingerprint generation to capture the device, vendor, and version for a particular firmware sample, and allow easier recognition of specific firmware (and thus an easy way to determine if a firmware is vulnerable). The final fingerprint is generated based on the filesystem of an image. After an analysis of 5296 images (retrieved using a web crawler), they note that the recall and precision of firmware fingerprints exceeds 90%, i.e., that this is an effective method to recognize a particular version of firmware (and thus whether it may contain vulnerabilities).

Code searching is especially effective for LB firmware, because one can use existing standards to guide the search. For example, CryptoREX [44] uses known seven widely used cryptographic libraries, detected using the Buildroot tool, to detect misuse of cryptographic APIs, which may have security implications. They track taint statically backwards from the call sites of the APIs, and if this tracking uncovers cryptographic function misuse based on OWASP guidelines, an alert is triggered. With 165 predefined cryptographic APIs, they discovered 679 issues in 1327 firmware images.

However, using known APIs for code search is not necessary: FirmUp [9] finds CVEs in stripped firmware images, by computing similarity between a query and target procedure. Each procedure is represented as hashed “strands” (subblocks of basic blocks that compute single outputs). They discover 373 vulnerabilities in around 2000 executables (out of 5000 publicly available firmware retrieved from vendors), improving performance over the state of the art BinDiff [48] by an average of 45%.

So far, most studies have only attempted to find new vulnerabilities in firmware, but it is also worth investigating whether, once a vulnerability and its corresponding mitigation is known, the firmware is patched to protect against that vulnerability. Yu et al. [42] conducted an in-depth study on the adoption of common attack mitigations against memory corruptions, by measuring their presence in more than ten thousand LB firmware of deployed embedded devices, collected through web crawling of vendor websites. To identify user- and kernel-level mitigations used in the firmware, they use various heuristics specific to the mitigation; for example, for stack canaries, they search for the string printed when the stack check fails, and check that the function that references this string is called from other functions. Kernel-level mitigations can be identified based on the architecture, kernel version, and building configurations, which can be recovered from the `.config` file in the kernel, or inferred from string constants and functions indicating the presence of mitigations if the `.config` file is missing (function symbols are recovered via Vmlinux-to-ELF). The results of their evaluation on 10,685 images scraped from vendors (with 7977 Linux kernels) show that the majority of embedded devices did not implement user-space or kernel-level attack mitigations.

Dynamic analysis. All dynamic large-scale LB research in this study has focused on exploitation of firmware at runtime. Firmadyne [3] is the first scalable automated system to use dynamic analysis, and try to exfiltrate information and run known exploits on firmware. First, they launch the firmware in a “learning mode”, where they record system interactions with the network, and then they construct a matching environment for emulation. They perform three automated analyses: they check for publicly accessible web pages from the LAN interface of the image, they check unauthenticated Simple Network Management Protocol information for sensitive data, and execute 60 known exploits from the Metasploit framework on the device. The system detected 74 known vulnerabilities affecting 887 firmware images (retrieved via web crawling of 42 pre-selected vendors), across at least 89 products. However, it does not determine whether the detected vulnerabilities are exploitable from the Internet, and uses custom pre-built kernels, so it cannot be used to confirm vulnerabilities in kernels or kernel modules contained in the firmware.

Subsequently, FirmAE [23] investigated the failure cases of Firmadyne [3], and found out that in the widespread case of discrepancies between the real and virtual environment, failures can often be avoided by simple heuristics. The authors proposed a technique of arbitrated emulation: instead of strictly following the original execution of firmware, at certain points they inject proper interventions to high-level behavior (e.g., network configuration or boot sequence), in order to allow analysis to continue. They implemented the prototype FirmAE, and evaluated it on a dataset of 1124 images download from vendor websites. Fuzzing the web services of firmware and running known exploits, FirmAE was able to detect 4.8 times more vulnerabilities than FIRMADYNE, showing that this is a viable way to increase the amount of firmware that can be successfully emulated. However, their heuristics are developed empirically based on the failures of FIRMADYNE, so they can only handle the observed cases, and may not apply to new devices and configurations.

Costin et al. [8] focus even more on web penetration: they run the firmware in an emulator, and use web penetration tools to analyze its security at runtime, computing differences in the emulated filesystem at different points in time. They specifically select firmware containing web server binaries, web configuration files, and server- or client-side web interface code, based on file names and contents (e.g., HTML). They evaluate the firmware for the presence of vulnerabilities such as command execution and cross-site scripting, and they check how many provide HTTPS support. They applied this framework to study the security of embedded web interfaces in COTS devices, and out of 246 firmware images that they could emulate (from an original dataset of 1925 retrieved from vendors), they found serious vulnerabilities in 185 (around 75%).

Since fuzzing may bring the device into an inconsistent state, such analysis could require manual intervention, rendering fuzzing infeasible on a large scale. To be able to conduct large-scale fuzzing analysis, FirmFuzz [36] uses a "snapshot and rollback" strategy to automatically revert firmware to a stable state if it becomes inconsistent. Hardware models are created automatically, by monitoring panic logs generated by a custom kernel when the firmware attempts to access a nonexistent peripheral. When testing for four types of vulnerabilities in 6427 firmware images scraped from 3 vendor websites, and fuzzing 'interesting' execution paths found through static taint analysis, they discover seven new vulnerabilities across six different devices.

4.3.2 Small-scale studies of LB firmware. There have also been several studies that evaluate fewer than 500 firmware samples [5, 17, 19, 39, 41, 46].

Static analysis. As an example of static analysis on a smaller scale, the Binary Analysis Tool [19] searches for strings, compares data compression, and computes differences between binaries. Given a binary, it uses these techniques to detect cloning of code from some repository of packages, with a primary rationale of detecting code violating the GNU General Public License. The authors evaluate the tool on a "small number" of firmware binaries (they explicitly list 2 in the paper), finding that it was able to detect cloning, albeit with false positives.

Another option is to conduct code search on a higher level, using a classifier to detect functionality and then categorize firmware based on a functionality profile. HumIDIFy [39] trains on a set of binaries, looking for features such as strings and function imports/exports, and the result of a classification of the testing set is a category label, such as 'web-server' or 'secure-shell daemon', and a confidence value. Based on the label, the system chooses a pre-built functionality profile, and when a binary does not follow that profile based on static analysis, there is potential unexpected or hidden functionality. The authors manually analyze 100 images scraped from vendor websites to create an initial set for training. On the rest of the dataset, they reach a classification accuracy of 96.45%, but a limitation is that this expected functionality profile must be defined manually.

It's possible to statically detect points where an attacker could compromise the execution of firmware: SaTC [5] uses taint tracking to detect security vulnerabilities in web services provided by embedded devices. The system recognizes front-end files and back-end programs based on file types (e.g., HTML for front-end and executable binaries for back-end), and analyzes the front-end files to extract potential keywords of user input (e.g., *deviceName*). Then, the system detects the border binaries in the back-end by matching the strings they contain against the extracted keywords, and tries to locate the points where the binaries receive user input, using taint analysis to track where untrusted data may be used. If it finds a vulnerable use of tainted data, such as in a system call parameter, it checks reachability of the use, and when it's reachable, an alert is raised. When evaluating it on 39 firmware samples collected from vendors, the authors discovered 33 unknown bugs, which is significantly more than previous work [33].

Dynamic analysis. LB firmware has an OS abstraction, and as such can be emulated in both system-mode and user-mode. Emulators such as QEMU often run firmware in system-mode, but user-mode execution is much faster, because there's no need for, e.g., address space translation or complete emulation of system calls. FIRM-AFL [46] combines the two methods in "augmented process emulation". Initially, the firmware boots in a system-mode emulator and the user-level programs launch inside it, and once the desired program is at a predetermined point (e.g., the beginning of the main function), it is migrated to user-mode emulation to improve execution speed. Re-using the dataset from Firmadyne [3], they found 288 firmware samples useful for analysis. In those, they observe, on average, an 8.2 times higher throughput compared to system-mode emulation-based fuzzing.

Some functions are more vulnerable to exploitation through unsafe input than others, and FIRMCORN [17] uses this knowledge to direct its fuzzing effort. First, the authors group functions in IoT firmware by complexity, and then rank them by vulnerability (a function is more vulnerable if it performs memory operations and uses sensitive functions). They then set the fuzzing entry point based on the vulnerability results, checking stack data after a sensitive function call to detect crashes. In evaluation on 10 samples obtained from the vendor, they achieve significantly higher throughput than the state-of-the-art [4], and discover two zero-day vulnerabilities.

Firmware in IoT devices often uses stateful protocols, such as *smb* and *ftp*, and to fuzz such a protocol, the fuzzer must set the system to the desired state in advance. IoTHunter [41] combines a message fuzzer and a state fuzzer: the message fuzzer generates new test cases to send as input to the firmware, and the state fuzzer keeps track of the state sequence and schedules protocol states. Evaluating on a dataset of 8 samples obtained from vendors, the authors observe that this solution outperforms the black-box fuzzer boofuzz [30] on the same dataset by up to 2.5× coverage, and finds five new vulnerabilities.

5 RESULTS

In total, we collected 34 relevant sources of information that discuss analysis of firmware, enumerated in Section 4. Of these 34 texts, 11 focus only on NLB firmware, and 9 only on *both* LB and NLB firmware. There are also 14 sources that focus only on LB firmware. We present the full list and categorization of NLB firmware research in Table 1, and research on both types of firmware in Table 2. In Figure 1, we can see the research separated by firmware type and the number of firmware samples they analyze: the number of samples is on the horizontal axis, the number of research papers is on the vertical axis, and the type of firmware analyzed in those studies is indicated by color. It is apparent that mostly LB research used a large amount of samples, while NLB research was conducted on a smaller scale.

We can see from Table 3 that the majority of studies examined less than 100 firmware samples. Specifically for NLB-only studies, only 2 analyzed more than 500 samples. In contrast, in the sources that we found discussing only LB analysis, there is a tendency for a larger number of samples.

The last row of Table 3 shows the firmware type focus of large-scale studies (more than 500 samples). It seems that in the history of embedded firmware research, only 8 analyses of LB firmware were conducted at large scale. It is clear that there is comparatively a lack of large-scale studies focusing only on NLB firmware – the majority of those analyzing a large number of samples either focus only on LB firmware, or a combination of both LB and NLB firmware. Furthermore, the majority of studies did not share their data; however, most of the large-scale studies either provided their dataset directly (or are planning to), released it partially, or provided a link to where images were obtained.

5.1 Non-Linux-based (NLB) firmware

Surveying existing analysis of NLB firmware, we observe that so far there have only been 2 large-scale studies of solely NLB firmware [16, 40], and 4 studies combining LB and NLB firmware [7, 14, 34, 45]. Those combining both types of firmware generally have a higher proportion of LB firmware (75-90%), though Shoshitaishvili et al. [35] use more NLB firmware. In addition, there have been several smaller-scale studies of only NLB firmware [6, 10, 13, 18, 21, 32, 43] and both NLB and LB firmware [12, 27, 31, 33, 35]. Where the majority of LB studies discussed in Section 4.3 used vendor websites for firmware retrieval, NLB studies use a wider variety of techniques, such as direct firmware extraction, custom web search, and mobile app-based crawling.

We see in Table 4 that the majority of NLB firmware analysis is focused on the ARM architecture, with MIPS in the second place; usually only 1 architecture is analyzed. For LB firmware, the majority of studies obtain samples from the vendors' websites; for NLB firmware, the approaches are more varied, including custom web searches, submission by users, direct firmware extraction from devices, manual compilation of firmware, and methods using mobile app stores. Table 5 shows the types of NLB devices that are examined in the research that we discuss here. IoT devices are the most common, followed by computer peripherals, and then personal, industrial, and other devices.

Furthermore, Table 6 shows which types of analyses have been the focus of NLB studies. It is clear that most research has been in the area of static code analysis, perhaps because it is the 'simplest' and fastest of code analyses; a disadvantage is that vulnerabilities found with static analysis often cannot be confirmed [31]. Static code analysis is followed in frequency by dynamic analysis and symbolic execution of code (we consider

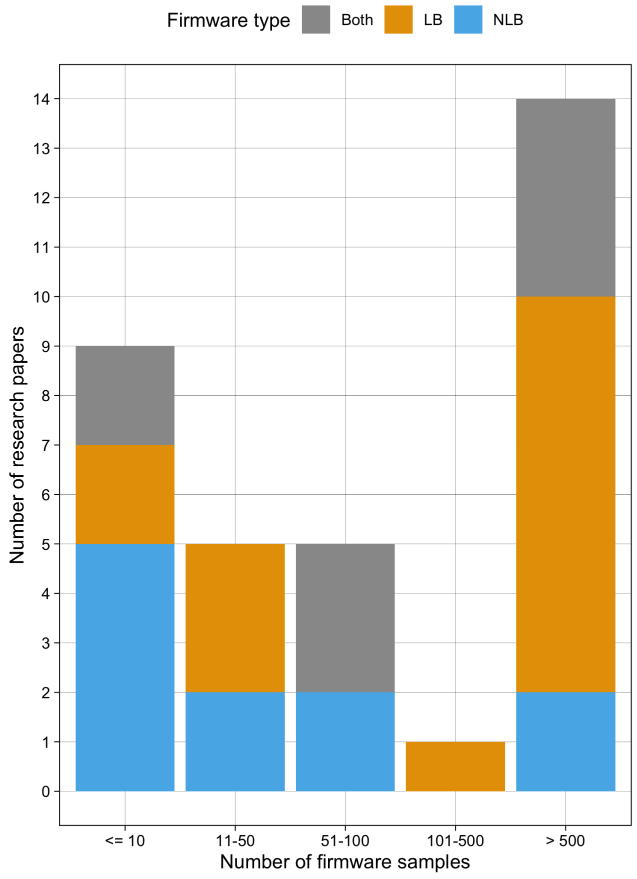

Figure 1: ***Past firmware research by type and number of samples.***

Document	Sector	Dataset status	ISAs	# Analyzed	# Scraped	Analysis focus	Scraping approaches	Analysis types
Gritti et al., [16]	personal, other, IoT, medical, industrial	linked	ARM	804	804	code	existing dataset	static, symbolic execution
Wen et al., [40]	personal, IoT, medical, other	available direct	ARM	793	793	code, config	app store	static
Davidson et al., [10]	other, computer peripherals	unavailable	RISC MSP430	99	99	interfaces	vendor website, custom search	symbolic execution
Feng et al., [13]	other, industrial	available direct	ARM, AVR, MIPS, RISC V	80	80	interfaces	vendor website	dynamic
Fowze et al., [15]	computer peripherals	unavailable	RISC MSP430, Intel 8051	29	29	code	vendor website	symbolic execution
Clements et al., [6]	other	unavailable	ARM	16	16	interfaces, code	own generation	dynamic
Zhu et al., [47]	industrial	unavailable	ARM	10	10	code	vendor website	static
Gustafson et al., [18]	IoT	available direct	ARM	6	6	code	vendor website, own generation	dynamic
Redini et al., [32]	personal	available direct	ARM	5	5	code	vendor website	static, dynamic, symbolic execution, taint
Zaddach et al., [43]	computer peripherals, IoT, personal	available direct	ARM	3	3	code	direct approach	dynamic, symbolic execution
Hernandez et al., [21]	computer peripherals	available partial	Intel 8051	2	2	code	vendor website	static, symbolic execution

Table 1: Categorisation of past research analyzing non-Linux firmware.

Document	Sector	Dataset status	ISAs	# Analyzed	# Scraped	Analysis focus	Scraping approaches	Analysis types
Zhao et al., [45]	IoT, industrial	available direct	ARM, Intel x86, MIPS	32817	34136	code	user submission, FTP, vendor website	static
Costin et al., [7]	personal, IoT, industrial	available direct	ARM, MIPS	32356	172751	credentials, config	vendor website, FTP, custom search, user submission	static
Feng et al., [14]	IoT	linked	ARM, MIPS, Intel x86	8126	33045	code	vendor website, FTP, existing dataset, own generation	static
Shirani et al., [34]	IoT	unavailable	ARM	5756	-	code	vendor website, FTP	static
Eschweiler et al., [12]	IoT	unavailable	Intel x86, Intel x64, ARM, MIPS	62	62	code	vendor website, existing dataset	static
Pewny et al., [31]	IoT	unavailable	ARM, Intel x86, MIPS	60	60	code	unknown	static
Redini et al., [33]	IoT, personal	available direct	ARM, AARCH64, PowerPC	53	952	code	vendor website, existing dataset	static, taint
Muench et al., [27]	other, IoT	available direct	ARM	4	4	code, interfaces	unknown	dynamic
Shoshitaishvili et al., [35]	IoT	unavailable	ARM, PowerPC	3	3	code	unknown	static, symbolic execution

Table 2: Categorisation of past research analyzing both LB and NLB firmware.

Firmware samples	LB	NLB	Both	Total
<= 10	2	5	2	9
11-50	3	2	0	5
51-100	0	2	3	5
101-500	1	0	0	1
> 500	8	2	4	14

Table 3: LB and NLB firmware studies by number of samples (> 500 is considered large-scale).

Device sector	Count
IoT	13
Computer peripherals	4
Industrial	5
Medical	2
Other	6
Personal	6

Table 5: Number of NLB studies by device sector.

Architecture	Count
AARCH64	1
ARM	17
AVR	1
Intel-8051	2
Intel-x64	1
Intel-x86	4
MIPS	6
PowerPC	2
RISC-MSP430	2
RISC-V	1

Table 4: Number of NLB firmware studies by CPU architecture.

Analysis focus	Static	Dynamic	Symb. exec.	Taint
Code	12	5	6	2
Interfaces	0	3	1	0
Configuration	2	0	0	0
Credentials	1	0	0	0

Table 6: Number of studies of NLB firmware by analysis methods and analysis focus.

these as two separate categories, because symbolic execution is a hybrid between static and dynamic execution). It makes sense that interfaces have not been analyzed statically, as they will only be active at runtime. There are two studies that implement taint analysis of code: static [33], and based on dynamic symbolic execution [32]. One area that seems not to have been explored much for NLB firmware is dynamic or symbolic analysis of configuration.

6 DISCUSSION

We have provided an overview of past studies on the topic of firmware research, particularly focusing on NLB firmware. In this section, we will comment on the significance of these findings, and what they mean for further research in this domain.

Historically, there has been much more research focused on LB firmware than on NLB firmware, which means that a comprehensive review of LB firmware research is not feasible in the scope of this study, Thus, from the outset, we must lean more towards searching for literature discussing NLB firmware, so our corpus is limited regarding LB firmware research, making it more difficult to draw comparisons between the two. However, even with the goal of specifically looking for NLB firmware research, we found many articles discussing only LB firmware, as summarized in Section 4.3. This shows the prevalence of research focusing on LB firmware, and supports our conclusions that it would be beneficial to expand research in NLB firmware.

For studies including NLB firmware, the dataset is frequently available. However, this could be improved, as for 8 out of 20 studies, the dataset was not available; this hinders the reproducibility of the results. Furthermore, the missing datasets add up to 6035 samples, which, were they to be made available, could be used in further research.

In existing studies, there is a bias in the datasets towards IoT firmware, perhaps because this type of firmware is easiest to scrape, as vendors make it available for retrieval from their servers. NLB research is more varied, probably because the IoT space has a majority of LB firmware; still, both of the large-scale NLB studies included in this paper contain IoT firmware. Therefore, NLB firmware retrieval at a large scale remains a challenge, particularly if the aim is to analyze firmware from multiple non-IoT sectors.

Furthermore, there is a bias towards ARM architectures in the datasets. The majority of embedded devices use the ARM architecture, so including ARM firmware is almost unavoidable with a large-scale study; however, all large-scale NLB studies analyze only ARM firmware [16, 40]. Yet, there is a sizable portion of devices that use other architectures, and limiting focus to only one architecture leads to the development of techniques specific to that architecture, leaving out other devices. Thus, there is also potential for research in analysis focused on non-ARM firmware, particularly at a large scale.

With NLB firmware, the biggest problem is the lack of standards, so one cannot know for certain the architecture or base address of a given sample. This results in obstacles preventing even basic static analysis of code. FirmXRay [40] solved the problem of architecture detection by using firmware from specific vendors, which has a known architecture, and HEAPSTER [16] re-used FirmXRay's dataset, adding 5 samples from a single vendor with a known architecture. For base address detection, FirmXRay developed a technique that worked for their dataset, and HEAPSTER applied the same technique. However, this technique is not generic: it relies on domain knowledge, namely vendor-specific functions in the firmware. The approach used by Zhu et al. [47] is more widely applicable, as it only relies on the presence of a sufficient amount of strings in the binary. Their method for finding string addresses uses ARM-specific byte sequence representations of `ldr` instructions, but instructions with the same function also exist in other architectures, so this part of the algorithm can easily be adapted by changing the target byte sequences in FIND-LDR. Nevertheless, the efficacy of this technique depends on the amount of ASCII strings in the binary, and how the compiler stores them, as one of the authors' assumptions is that strings used in adjacent code will be stored in a contiguous block of memory; this may not be true for all compilers or for hand-crafted assembly code. In general, past studies used firmware where the architecture was known before analysis, but in a large-scale study with a varied dataset, firmware will often not be known. Hence, another possible direction for further research is the development of techniques and processes that can analyze unknown firmware in a generic manner.

7 CONCLUSION

In this paper, we discussed existing research in the area of embedded device firmware analysis. We proposed a classification method where we separated studies by whether they analyzed only Linux-based firmware, only non-Linux-based firmware, or both. We then categorized them based on several other factors, such as the analysis methods they used and the number of firmware samples they analyzed. We found that the majority of large-scale studies focused mainly on Linux-based firmware, with a small number analyzing both. Most non-Linux-based studies examine firmware built for the ARM architecture, with prior knowledge of the target device specifications, and the most common device sector for both Linux-based and non-Linux-based firmware is the 'Internet of Things'. The results indicate that development of scalable generic analysis techniques for unknown non-Linux firmware could be a useful direction for future work.

REFERENCES

[1] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. 2017. Understanding the Mirai Botnet. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1093–1110. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis

[2] Pieter Arntz. 2022. ZuoRAT is a sophisticated malware that mainly targets SOHO routers. https://blog.malwarebytes.com/reports/2022/06/zuorat-is-a-sophisticated-malware-that-mainly-targets-soho-routers/.

[3] Daming Chen, Manuel Egele, Maverick Woo, and David Brumley. 2016. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. https://doi.org/10.14722/ndss.2016.23415

[4] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. Lau, M. Sun, R. Yang, and K. Zhang. 2018. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In *NDSS*.

[5] Libo Chen, Yanhao Wang, Quanpu Cai, Yunfan Zhan, Hong Hu, Jiaqi Linghu, Qinsheng Hou, Chao Zhang, Haixin Duan, and Zhi Xue. 2021. Sharing More and Checking Less: Leveraging Common Input Keywords to Detect Bugs in Embedded Systems. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 303–319. https://www.usenix.org/conference/usenixsecurity21/presentation/chen-libo

[6] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. 2020. HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1201–1218. https://www.usenix.org/conference/usenixsecurity20/presentation/clements

[7] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. 2014. A Large-Scale Analysis of the Security of Embedded Firmwares. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA,

95–110. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/costin

[8] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. 2016. Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces. In *The 11th ACM on Asia Conference on Computer and Communications Security*. 437–448. https://doi.org/10.1145/2897845.2897900

[9] Yaniv David, Nimrod Partush, and Eran Yahav. 2018. FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, Xipeng Shen, James Tuck, Ricardo Bianchini, and Vivek Sarkar (Eds.). ACM, 392–404. https://doi.org/10.1145/3173162.3177157

[10] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. 2013. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *22nd USENIX Security Symposium (USENIX Security 13)*. USENIX Association, Washington, D.C., 463–478. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/davidson

[11] Moritz Eckert, Antonio Bianchi, Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2018. HeapHopper: Bringing Bounded Model Checking to Heap Implementation Security. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 99–116. https://www.usenix.org/conference/usenixsecurity18/presentation/eckert

[12] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/discovre-efficient-cross-architecture-identification-bugs-binary-code.pdf

[13] Bo Feng, Alejandro Mera, and Long Lu. 2020. P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1237–1254. https://www.usenix.org/conference/usenixsecurity20/presentation/feng

[14] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable graph-based bug search for firmware images. In *CCS 2016 - Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (Proceedings of the ACM Conference on Computer and Communications Security)*. Association for Computing Machinery, 480–491. https://doi.org/10.1145/2976749.2978370

[15] Farhaan Fowze, Dave Tian, Grant Hernandez, Kevin Butler, and Tuba Yavuz. 2021. ProXray: Protocol Model Learning and Guided Firmware Analysis. *IEEE Transactions on Software Engineering* 47, 9 (2021), 1907–1928. https://doi.org/10.1109/TSE.2019.2939526

[16] Fabio Gritti, Fabio Pagani, Ilya Grishchenko, Lukas Dresel, Nilo Redini, Christopher Kruegel, and Giovanni Vigna. 2022. HEAPSTER: Analyzing the Security of Dynamic Allocators for Monolithic Firmware Images. In *Proceedings of the IEEE Symposium on Security & Privacy (S&P)*.

[17] Zhijie Gui, Hui Shu, Fei Kang, and Xiaobing Xiong. 2020. FIRMCORN: Vulnerability-Oriented Fuzzing of IoT Firmware via Optimized Virtual Execution. *IEEE Access* 8 (2020), 29826–29841. https://doi.org/10.1109/ACCESS.2020.2973043

[18] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, and Giovanni Vigna. 2019. Toward the Analysis of Embedded Firmware through Automated Re-hosting. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. USENIX Association, Chaoyang District, Beijing, 135–150. https://www.usenix.org/conference/raid2019/presentation/gustafson

[19] Armijn Hemel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. 2011. Finding software license violations through binary code clone detection. In *MSR '11*.

[20] Martin Herfurt. 2022. Project TEMPA. https://trifinite.org/stuff/project_tempa/.

[21] Grant Hernandez, Farhaan Fowze, Dave (Jing) Tian, Tuba Yavuz, and Kevin R.B. Butler. 2017. FirmUSB. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM. https://doi.org/10.1145/3133956.3134050

[22] IEEE. 1983. IEEE Standard Glossary of Software Engineering Terminology. *ANSI/ IEEE Std 729-1983* (1983), 1–40. https://doi.org/10.1109/IEEESTD.1983.7435207

[23] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. 2020. FirmAE: Towards Large-Scale Emulation of IoT Firmware for Dynamic Analysis. In *Annual Computer Security Applications Conference* (Austin, USA) *(ACSAC '20)*. Association for Computing Machinery, New York, NY, USA, 733–745. https://doi.org/10.1145/3427228.3427294

[24] Constantinos Kolias, Georgios Kambourakis, Angelos Stavrou, and Jeffrey Voas. 2017. DDoS in the IoT: Mirai and Other Botnets. *Computer* 50, 7 (2017), 80–84. https://doi.org/10.1109/MC.2017.201

[25] Qiang Li, Xuan Feng, Raining Wang, Zhi Li, and Limin Sun. 2018. Towards Fine-grained Fingerprinting of Firmware in Online Embedded Devices. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*. 2537–2545. https://doi.org/10.1109/INFOCOM.2018.8486326

[26] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. [n.d.]. This POODLE Bites: Exploiting The SSL 3.0 Fallback. https://www.openssl.org/~bodo/ssl-poodle.pdf

[27] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. 2018. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS 2018, Network and Distributed Systems Security Symposium, 18-21 February 2018, San Diego, CA, USA*, ISOC (Ed.). San Diego. © ISOC. Personal use of this material is permitted. The definitive version of this paper was published in NDSS 2018, Network and Distributed Systems Security Symposium, 18-21 February 2018, San Diego, CA, USA and is available at : http://dx.doi.org/10.14722/NDSS.2018.23166.

[28] Karsten Nohl, Sascha Krissler, and Jakob Lell. [n.d.]. BadUSB — On accessories that turn evil. https://web.archive.org/web/20161019034729/https://srlabs.de/wp-content/uploads/2014/07/SRLabs-BadUSB-BlackHat-v1.pdf

[29] Amy Nordrum. 2016. The internet of fewer things [News]. *IEEE Spectrum* 53, 10 (2016), 12–13. https://doi.org/10.1109/MSPEC.2016.7572524

[30] Joshua Pereyda. [n.d.]. Boofuzz: Network Protocol Fuzzing for Humans. https://github.com/jtpereyda/boofuzz.

[31] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-Architecture Bug Search in Binary Executables. In *2015 IEEE Symposium on Security and Privacy*. 709–724. https://doi.org/10.1109/SP.2015.49

[32] Nilo Redini, Aravind Machiry, Dipanjan Das, Yanick Fratantonio, Antonio Bianchi, Eric Gustafson, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2017. BootStomp: On the Security of Bootloaders in Mobile Devices. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 781–798. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/redini

[33] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2020. Karonte: Detecting Insecure Multi-binary Interactions in Embedded Firmware. 1544–1561. https://doi.org/10.1109/SP40000.2020.00036

[34] Paria Shirani, Leo Collard, Basile L. Agba, Bernard Lebel, Mourad Debbabi, Lingyu Wang, and Aiman Hanna. 2018. BINARM: Scalable and Efficient Detection of Vulnerabilities in Firmware Images of Intelligent Electronic Devices. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 15th International Conference, DIMVA 2018, Saclay, France, June 28-29, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10885)*, Cristiano Giuffrida, Sébastien Bardin, and Gregory Blanc (Eds.). Springer, 114–138. https://doi.org/10.1007/978-3-319-93411-2_6

[35] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society. https://www.ndss-symposium.org/ndss2015/firmalice-automatic-detection-authentication-bypass-vulnerabilities-binary-firmware

[36] Prashast Srivastava, Hui Peng, Jiahao Li, Hamed Okhravi, Howard E. Shrobe, and Mathias Payer. 2019. FirmFuzz: Automated IoT Firmware Introspection and Analysis. *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things* (11 2019), 15–21. https://doi.org/10.1145/3338507.3358616

[37] Inc. Synopsys. [n.d.]. The Heartbleed Bug. https://heartbleed.com/

[38] The Linux Kernel Organization. [n.d.]. The Linux Kernel. https://kernel.org/

[39] Sam Thomas, Flavio D. Garcia, and Tom Chothia. 2017. HumIDIFy: A Tool for Hidden Functionality Detection in Firmware. In *14th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA '17), Proceedings (Lecture Notes in Computer Science)*. Springer, 279–300. https://doi.org/10.1007/978-3-319-60876-1_13

[40] Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. 2020. FirmXRay: Detecting Bluetooth Link Layer Vulnerabilities From Bare-Metal Firmware. *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Oct 2020). https://doi.org/10.1145/3372297.3423344

[41] Bo Yu, Pengfei Wang, Tai Yue, and Yong Tang. 2019. Poster: Fuzzing IoT Firmware via Multi-stage Message Generation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 2525–2527. https://doi.org/10.1145/3319535.3363247

[42] Ruotong Yu, Francesca Del Nin, Yuchen Zhang, Shan Huang, Pallavi Kaliyar, Sarah Zakto, Mauro Conti, Georgios Portokalidis, and Jun Xu. 2022. Building Embedded Systems Like It's 1996. (2022). https://doi.org/10.48550/ARXIV.2203.06834

[43] Jonas Zaddach, Luca Bruno, Aurélien Francillon, and Davide Balzarotti. 2014. Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. https://doi.org/10.14722/ndss.2014.23229

[44] Li Zhang, Jiongyi Chen, Wenrui Diao, Shanqing Guo, Jian Weng, and Kehuan Zhang. 2019. CryptoREX: Large-scale Analysis of Cryptographic Misuse in IoT Devices. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. USENIX Association, Chaoyang District, Beijing, 151–164. https://www.usenix.org/conference/raid2019/presentation/zhang-li

[45] Binbin Zhao, Shouling Ji, Jiacheng Xu, Yuan Tian, Qiuyang Wei, Qinying Wang, Chenyang Lyu, Xuhong Zhang, Changting Lin, Jingzheng Wu, and Raheem Beyah. 2022. A Large-Scale Empirical Analysis of the Vulnerabilities Introduced by Third-Party Components in IoT Firmware. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 442–454. https://doi.org/10.1145/3533767.3534366

[46] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. 2019. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1099–1114. https://www.usenix.org/conference/usenixsecurity19/presentation/zheng

[47] Ruijin Zhu, Baofeng Zhang, Junjie Mao, Quanxin Zhang, and Yu an Tan. 2017. A methodology for determining the image base of ARM-based industrial control system firmware. *International Journal of Critical Infrastructure Protection* 16 (2017), 26–35. https://doi.org/10.1016/j.ijcip.2016.12.002

[48] Zynamics. [n.d.]. BinDiff. https://www.zynamics.com/bindiff.html.