HALE-IoT: Hardening Legacy Internet of Things Devices by Retrofitting Defensive Firmware Modifications and Implants

Javier Carrillo-Mondéjar, Hannu Turtiainen, Andrei Costin, Jose Luis Martínez, and Guillermo Suarez-Tangil

***Abstract*—Internet of Things (IoT) devices and their firmware are notorious for their lifelong vulnerabilities. As device infection increases, vendors also fail to release patches at a competitive pace. Despite security in IoT being an active area of research, prior work has mainly focused on vulnerability detection and exploitation, threat modeling, and protocol security. However, these methods are ineffective in preventing attacks against legacy and End-Of-Life devices that are already vulnerable. Current research mainly focuses on implementing and demonstrating the potential of malicious modifications. Hardening emerges as an effective solution to provide IoT devices with an additional layer of defense. In this article, we bridge these gaps through the design of *HALE-IoT*, a generically applicable systematic approach to HArdening LEgacy IoT non-low-end devices by retrofitting defensive firmware modifications *without access to the original source code*. *HALE-IoT* approaches this nontrivial task via binary firmware reversing and modification while being underpinned by a semiautomated toolset that aims to keep cybersecurity of such devices in a *hale* state. Our focus is on both modern and, especially, legacy or obsolete IoT devices as they become increasingly prevalent. To evaluate the effectiveness and efficiency of *HALE-IoT*, we apply it to a wide range of IoT devices by retrofitting 395 firmware images with defensive implants containing an intrusion prevention system in the form of a Web Application Firewall (for prevention of Web-attack vectors), and an HTTPS-proxy (for latest and full end-to-end HTTPS support) using emulation. We also test our approach on four physical devices, where we show that *HALE-IoT* successfully runs on protected and quite constrained devices with as low as 32 MB of RAM and 8 MB of storage. Overall, in our evaluation, we achieve good performance and reliability with a remarkably accurate detection and prevention rate for attacks coming from both real CVEs and synthetic exploits.**

***Index Terms*—Cybersecurity, defensive techniques, devices, end-of-life (EOL), firmware, firmware modification, HTTPS, Internet of Things (IoT), legacy, retrofit security, secure socket layer (SSL)-proxy, Web application firewall (WAF).**

I. Introduction

INTERNET of Things (IoT) devices have notoriously vulnerable firmware [1], [2], [3]. Exploiting these vulnerabilities is often trivial, an example being the case of the infamous Mirai botnet [4]. Unfortunately, keeping the firmware of these devices updated is challenging. First, in many cases, a firmware update or a patch is simply not available. This is a prevalent problem due to the number of legacy devices connected to the Internet [4], [5], [6], [7], [8], [9]. Second, firmware is built-in to the devices, while automated—Over-The-Air (OTA)—firmware updates are generally not implemented or still have limited adoption [10]. Updating IoT devices, when and if available, may require fairly technical manual intervention, including having admin access and reflashing the device, and can prove challenging and error prone even for experienced users. These difficulties foster a culture of bad security hygiene around IoT. As a result, many IoT devices are left vulnerable, with dire and long-lasting consequences [11]. For instance, researchers recorded over 1.5 billion attacks against IoT devices in the first half of 2021 [12]. In this context, just one single vulnerability (CVE-2021-28372) [13] affected around 83 million devices, while some others (e.g., CVE-2013-7471) have been active for years and are still seen in the wild.[1]

In the absence of regular updates, bastioning IoT devices and hardening potentially vulnerable services emerge as first-line defense strategies. The Center for Internet Security (CIS) offers prehardened images and hardening checklists that have

Manuscript received 31 March 2022; revised 23 June 2022, 22 August 2022, and 11 October 2022; accepted 10 November 2022. Date of publication 24 November 2022; date of current version 9 May 2023. This work supported by the "Decision of the Research Dean on Research Funding (20.04.2022)" and "Dean's Decision on Researcher Mobility Grants (17.1.2022)" within the Faculty of Information Technology of University of Jyväskylä. The work of Javier Carrillo-Mondéjar was supported in part by the Spanish Ministry of Economic Affairs and Digital Transformation under Project RTI2018-098156-B-C52; in part by the Spanish Ministry of Science and Innovation under Project PID2021-123627OB-C52; in part by the Regional Government of Castilla-La Mancha under Project SBPLY/17/180501/000353 and Project SBPLY/21/180501/000195; and in part by the Spanish Education, Culture and Sports Ministry under Grant FPU 17/03105. The work of Guillermo Suarez-Tangil was supported in part by the "Ramon y Cajal" Fellowship through MCIN/AEI/10.13039/501100011033 and ESF "The European Social Fund Invests in Your Future" under Grant RYC-2020-029401-I, and in part by the Spanish Ministry of Science and Innovation through MCIN/AEI/10.13039/501100011033 and ESF "The European Social Fund Invests in Your Future" under COMET Grant TED2021-132900A-I00. *(Corresponding author: Andrei Costin.)*

Javier Carrillo-Mondéjar and Jose Luis Martínez are with the Albacete Research Institute of Informatics, University of Castilla-La Mancha, 02071 Albacete, Spain (e-mail: javier.carrillo@uclm.es; JoseLuis.Martinez@uclm.es).

Hannu Turtiainen and Andrei Costin are with the Faculty of Information Technology, University of Jyväskylä, 40100 Jyväskylä, Finland (e-mail: turthzu@jyu.fi; ancostin@jyu.fi).

Guillermo Suarez-Tangil is with the Cybersecurity Group, IMDEA Networks Institute, 28918 Madrid, Spain (e-mail: guillermo.suarez-tangil@imdea.org).

Digital Object Identifier 10.1109/JIOT.2022.3224649

[1] See timeline in: https://vuldb.com/?id.136365.

been adopted by the research community, which includes mechanisms to disable nonessential services [14]. While this reduces the attack surface, essential services may still suffer vulnerabilities [15]. Furthermore, existing approaches like [14], [16] do not address the constraints and the heterogeneity of modern IoT devices. Thus, applying off-the-land defenses at the network level, like third-party firewalls, has already been the subject of research [17], [18], [19]. The next line of defense includes retrofitting active [20], [21], and retroactive defenses [4], [22]. Retrofitting defenses into IoT devices offers the same advantages as general-purpose hardening, while enhancing their security mechanisms even without the support of the manufacturer.

Retrofitting security to legacy IoT devices faces many challenges. First, the firmware stock is large and heterogeneous, so nongeneric solutions hinder the adoption of this defense. Second, injecting externally compiled code and then expecting it to tightly co-exist with the firmware is a challenging and error prone process. Third, IoT devices generally have constrained resources and I/O interfaces, so they cannot easily accommodate arbitrary defensive solutions (e.g., intrusion detection system (IDS), antivirus) that are useful good defending traditional computing devices (e.g., PCs, laptops, and servers). Constrained by such challenges, existing approaches are limited in the scope of their implementation. For instance, Cui and Stolfo [20] presented a binary-patching tool called Doppelgänger that only offers in-practice protection against rootkits. Doppelgänger is essentially a memory integrity monitor that computes hashes of memory regions where "critical system processes" are mapped. The system then monitors changes in the hashes as a way to detect function hooking and other types of code injection. While Doppelgänger can compute and monitor hashes for any arbitrary memory region, identifying and understanding those regions requires considerable human expertise. Thus, approaches, such as Doppelgänger [20], do not scale in practice, and cannot be deployed systematically. Other works focus on hardening particular types of Web applications against cross-site scripting (XSS) and structured query language injection (SQLi) [15] attacks. However, these approaches rely on modifying the Web interpreters, which requires: 1) deep software modifications; 2) intimate knowledge of the targeted technology (e.g., PHP); 3) tedious preautomation taint annotations (e.g., sensitive sinks); and 4) access to source code.

To address existing shortcomings, we use a concept similar (yet somewhat distinctive) to symbiotic embedded machines (SEMs) [20] to design a systematic approach to hardening legacy *non-low-end IoT devices*. For clarity, we try to outline *HALE-IoT*'s encompassing definition of *non-low-end IoT device*. In the most general sense, at present *HALE-IoT* targets the *Type-I: General purpose OS-based devices* (e.g., Linksys EA6300v1 with Cortex A-9) as defined by Muench et al. [23] in their state-of-the-art work. Our distinction between *non-low-end* versus *low-end* devices is also generally in line with that in [23]. In other words, from a *HALE-IoT* perspective, *Type-II: Embedded OS-based devices* mapping to MCUs such as ARM7TDMI-S with flash memory in the range of 512-1024 kB and RAM memory 58–98 kB (e.g., Foscam FI8918W), and *Type-III: Devices without an OS-Abstraction* mapping to MCUs such as Cortex M-3 with flash memory in the range of 16–1024 kB and RAM memory 80–256 kB (e.g., STM Nucleo-L152RE), would qualify as *low-end* devices, and are therefore unsupported in present iteration of *HALE-IoT*. Certainly, the taxonomy of device types defined in [23] is one of many possible, and taxonomy definition is strongly influenced by the problem space at hand. However, to date, that of Muench et al. [23] best reflects the research perspective to which *HALE-IoT* applies. In addition, due to the binary and configuration sizes of *HALE-IoT* (Sections IV-C and V-E), we do not aim for devices with less than 32-MB RAM and less than 8-MB storage. A direct consequence of this is that microcontroller/MCU-based IoT devices (such as MSP430, ARM Cortex-M0, and similar ARM MCU families) are exclued at this stage of *HALE-IoT* development owing to their total/free RAM and storage limitations. Moreover, *HALE-IoT* currently works for Linux-derived or BusyBox-based systems, which almost by default excludes MCU-based IoT devices, as Linux is particularly challenging to scalably run on MCU devices, though an exception exists for MCUs running uClinux [23]. As we demonstrate later in this article, *HALE-IoT* successfully supports not only powerful boards such as Raspberry Pi, but an additional wide range of such "Type I" [23] devices, as depicted in Table IV. We show that *HALE-IoT* successfully runs on quite constrained devices with as low as 32 MB of RAM and 8 MB of storage, parameters that in our view absolutely qualify *HALE-IoT* to support a wide-range and large-numbers of commercial off the shelf (COTS) devices.

> With *HALE-IoT*, we essentially retrofit complex defense systems into raw firmware binaries via systematic yet minimally intrusive low-level modifications that *do not require access to the original source code*. Our approach differs from the state-of-the-art in several ways. First and foremost, we design a generically applicable framework to provide reliable security and protective standards to legacy firmware. Second, we develop a systematic testing methodology that constitutes the first benchmark to assess the effectiveness of retrofitting defensive firmware modifications.

We developed a cross-platform system, called *HALE-IoT*, that at the time of writing successfully runs on at least MIPSeb, MIPSel, ARMel, and Intel 80386 architectures. *HALE-IoT* incorporates several industry-standard security tools. We devised a battery of tests using real-world attacks, particularly focusing its evaluation on fuzzing the Web interface, for two main reasons. One is that the Web-interfaces are well known to be exposed and lacking security in many aspects [24], [25], while IoT devices are often proven to have their Web-interfaces highly vulnerable and exposed [2], [3]. Another reason is that, as several studies have reported, (I)IoT devices are much more often run missing, lax, or insecure secure socket layer (SSL)/transport layer security (TLS) implementations [26], [27], [28], [29], and make insignificant contributions to secure TLS [26].

We note that while our evaluation reports detection rates, its main focus is not to assess how well *HALE-IoT* detects and prevents real-world attacks. In essence, *HALE-IoT* embeds

industry-standard protection mechanisms such as have been widely tested before, e.g., Web application firewall (WAF) such as Raptor [30]. Our aim is to assess whether the retrofitted defensive mechanism can effectively (i.e., detect and protect) and correctly co-exist within the retrofitted firmware without preventing normal use of the system (e.g., not crashing it). This is important, as software projects such as full-fledged WAFs are fairly sophisticated. To the best of our knowledge, no prior work has attempted and assessed the feasibility of implanting sophisticated frameworks[2] into IoT firmware.

We evaluated the effectiveness of our methodology using 395 different firmware images from a wide range of vendors, including D-Link, Netgear, Linksys, TRENDnet, and OpenWrt. We emulate those 395 firmwares using a similar procedure as in the state-of-the-art works [2], [3]. Due to the difficulty of acquiring hardware for all vendors, we restrict our bare-metal evaluation to four physical devices (Table IV) featuring 32–1024 MB of RAM and 8–4096 MB of storage, while representing both ARM and MIPS architectures as well as open-source and proprietary hardware and firmware. At the same time, we note that our current efforts do not attempt to test *HALE-IoT* in the "long trail" of architectures (e.g., niche architectures and targets not supported by GCC,[3] nor on architectures that are supported but are not widespread, the same way we do not claim that *HALE-IoT* works on low-end and very constrained targets such as MSP430). A more extensive evaluation is part of future work.

Our main contributions are summarized as follows.

1) We develop a generic methodology supported by *a system architecture and a reference implementation* for hardening legacy IoT devices via defensive firmware retrofitting and implants. To the best of our knowledge, this is the first of its kind.
2) We evaluate the effectiveness of our methodology by *testing it on potentially vulnerable and insecure Web-interfaces* of a large and diverse set of IoT vendors and devices.
3) We identify and derive several core challenges of this problem space that require further attention and research.

The remainder of this article is organized as follows. We first present the overall applicable threat model in Section II. Then, we introduce and detail the *HALE-IoT* architecture and methodology in Section III. In Section IV, we detail the experimental setup, the data sets. We present the testing methodology and the results in Section V. We then discuss challenges and future improvements in Section VII. We present and discuss related work in Section VIII. Finally, we conclude this article with Section IX.

II. Threat Overview

There are millions of devices connected to the Internet that shape the way users interact with technology. These devices have many attractive features that make them popular. Unfortunately, many of these devices lack basic security and privacy protections. This leaves IoT devices exposed to major security issues ranging from insecure configurations and protocols (i.e., Telnet and HTTP) to outdated software with known vulnerabilities and public exploits.

From an attacker's point of view, IoT devices are very attractive due to the weaknesses they present and the absence of IoT-centered defensive tools (e.g., Antivirus, IDS. Mirai is a proof of this. Mirai, the first malware specifically designed to infect IoT managed to infect around 600 000 devices [4]. Unlike the early versions of Mirai, which used only a set of usernames and passwords to gain access to IoT devices via insecure Telnet and SSH configurations, IoT malware currently incorporates a wide portfolio of exploits for N-days vulnerabilities in order to gain access and install and spread their malware [31]. In particular, in a large number of cases they (ab)use CVEs for Web-interfaces [32], [33], [34].

IoT devices often require network management interfaces for their configuration and maintenance (i.e., Telnet, SSH, and HTTP), due to the lack of interactive interfaces like the ones offered in desktop computers (i.e., mouse, keyboard, video). Consequently, these network services are exposed to attackers, causing well-known security issues, as shown in [2], [3], and [35]. Costin et al. [2] performed a large-scale analysis of Web services provided by different IoT devices, discovering 225 high-impact vulnerabilities (i.e., Command execution, XSS) verified through dynamic analysis, and around 9000 possible vulnerabilities reported through static analysis in 185 firmware images that were analyzed. These security issues, coupled with shortage of security updates or patches, make IoT devices an attractive target for attackers, allowing them to create large botnets or to mine cryptocurrency [11], [36].

A. Threat Model

Because of *HALE-IoT*'s architecture and the evaluations performed (i.e., HTTP and WAF), when building our threat model we reference the generalized threat model for WAFs, based on the extensive state-of-the-art survey by Li and Xue [37]: 1) the Web application itself is benign (i.e., not hosted or owned for malicious purposes) and hosted on a trusted and hardened infrastructure (i.e., a trusted computing base, including OS, Web server, interpreter, etc.) and 2) the attacker is able to manipulate either the contents or the sequence of Web requests sent to the Web application but cannot directly compromise the infrastructure or the application code.

Therefore, the overall threat model of *HALE-IoT* could be seen as a generalized form of the threat model by Li and Xue [37], and could be summarized as follows.

1) The entire firmware and underlying OS is benign (i.e., not hosted or owned for malicious purposes) and hosted on an original, trusted, or hardened device or infrastructure (i.e., a trusted computing base, including OS, Web server, interpreter, etc.).
2) The attacker is able to manipulate either the contents or the sequence of network requests (e.g., HTTP requests, FTP and Telnet commands, lower-level network packets) sent to the device firmwares but cannot or did not directly

[2]For instance, at the time of writing, Raptor has an estimated 22700 LoC, and SSL-proxy with Golang has an estimated 8600 LoC. Raptor and Golang are two of the frameworks we systematically retrofit.

[3]https://blog.yossarian.net/2021/02/28/Weird-architectures-werent-supported-to-begin-with

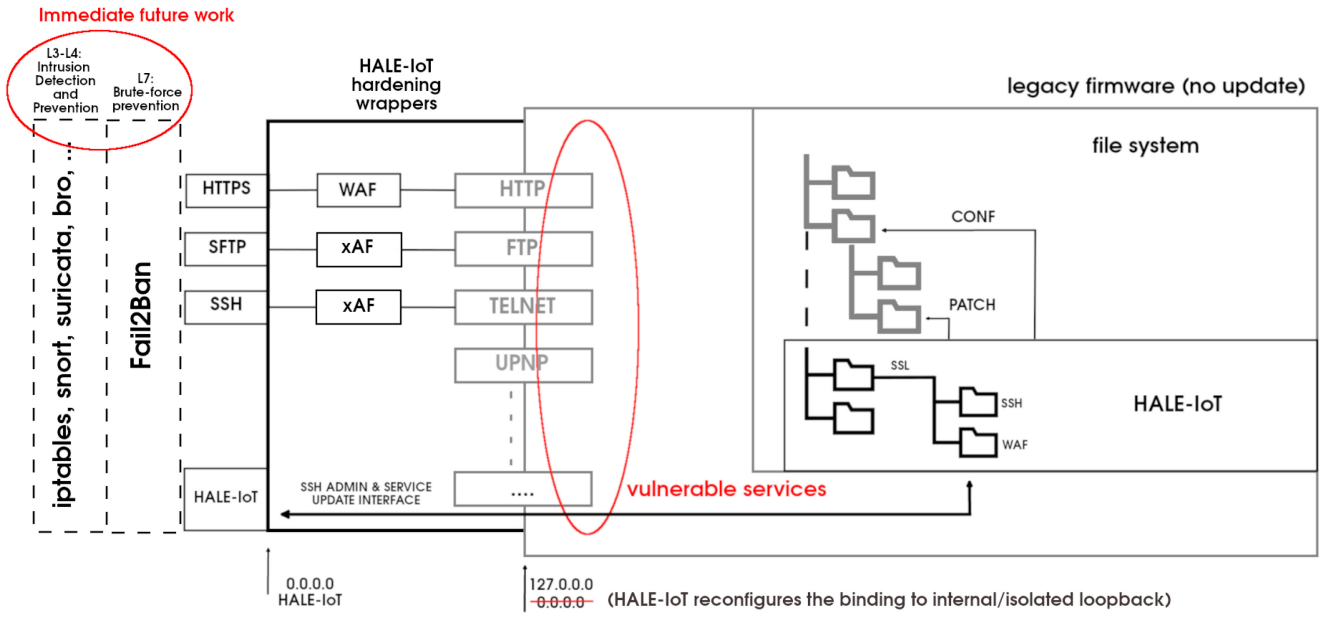

Fig. 1. IoT firmware system diagram after hardening with *HALE-IoT* (shows also long-term vision for additional hardening at layers L3–L4, L7).

compromise the infrastructure or the application/firmware code prior to connecting the device to the Internet or infrastructure (namely, supply chain attacks).

Focusing on the threat model in more detail, we consider a legacy device with Linux-derived or BusyBox-based firmware with network services (e.g., HTTP, FTP) that are vulnerable to known attacks publicly available on the Internet. We consider N-day vulnerabilities because they are the ones that primarily endanger the security of IoT devices [38] and the ones that are easily exploitable with the help of IoT device scanners [35]. Our threat model can also consider zero-day attacks, as long as they only target hardened services that no longer become exposed to the Internet after applying *HALE-IoT*'s methodology (e.g., Telnet). Our work is focused on hardening legacy systems for which there are no security updates from the manufacturer.

To harden IoT devices via defensive firmware retrofitting, we proceed with the following assumptions. First, we have access to the firmware image of the device (e.g., downloaded from Internet [1], extracted from device [39]). This allows us to modify the file system and update the device with the hardened-modified firmware version, though there are exceptions when we do not even require access to the firmware nor firmware's modification (Section VI-E). Second, there is storage on the device to add new binaries that will take care of listening on the service's original interface and port, so that it wraps everything around a WAF to its original service. This allows the original functionality of the device to remain unaffected. The security of the WAF is out of the scope of this article, as it is both research-wide and industry-wide accepted methods for securing Web-apps [37]. Therefore, we assume that adding a WAF as part of the *HALE-IoT* methodology increases device security. (We discuss the limitations of WAFs in Section VIII-D.) Third, it is possible to reconfigure the original Web server through configuration files or scripts, that is, we can change the interface and port in which the original service is bound. Fourth, we consider vulnerabilities that can be triggered through a Web request. The type of vulnerability exploited will belong to the category of errors in user input validation [37], to which correspond vulnerabilities involving SQL injection, CIs and path traversal, among others. Finally, we assume that new vulnerabilities will appear in the future and that the rules or software will need to be updated.

With these assumptions in mind, the next section introduces *HALE-IoT*, a methodology designed to offer an extra layer of security to IoT devices, focusing on the protection of the core and the most vulnerable services exposed to the Internet by IoT devices.

III. HALE-IoT

This section introduces the main components of our system.

A. Methodology & System Overview

HALE-IoT is a system designed to harden IoT devices. With the wide spread network of obsolete IoT devices in mind, our system focuses on legacy and end-of-life (EOL) firmware for the following reasons: devices with legacy firmware: 1) are prone to be vulnerable; 2) are extremely likely to expose exploitable services leading directly to breaches; and 3) will almost certainly never get a security patch. In Fig. 1, we depict a high-level view of a hardened IoT firmware/device. From the point of view of firmware's "inner components," the diagram captures two dimensions: 1) "vulnerable services"—the services that will eventually be spawned when loading the firmware (e.g., a Web HTTP server, Telnet, and FTP) and 2) "file system"—the original file system itself, where the binaries and the configuration of such potentially vulnerable services reside. The diagram also represents the entry point of

these services, typically a network interface. A service can listen to different interfaces at the same time (e.g., Wireless or Wide/Local Area Network, WLAN and WAN/LAN, respectively). We represent as 0.0.0.0 a generic network interface that is very likely to be attacker-accessible.

Methodology: We follow three core principles that underpin the development methodology of *HALE-IoT*. First, the hardening process has to be generic and flexible to accommodate the most popular services available in IoT devices. We also require that the system can accept the integration of generic protection mechanisms that match in complexity the type of attacks that generally target IoT devices. Second, we follow the fail-safe minimization principle [40] by which the modifications we introduce during the hardening process should be as unintrusive as possible, always preserving the normal operation of the device. In other words, *HALE-IoT* will make minimal changes to the firmware, having its main focus first hardening the system via reconfiguration, then patching existing *configuration files*, only proceeding to make code-level modifications (namely, binary patching) as a last resort. Only in situations when binary patching is necessary, we apply a twofold strategy: the analysis phase—a human-guided semi-automatic process that produces a proof-of-concept; and the deployment phase—which can reproduce the patching and retrofitting at scale in a fully automated fashion.

System design: HALE-IoT leverages the methodology above to design a practical system that addresses the challenge of hardening heterogeneous devices from the following angles.

1) *Secure Frontend:* This step aims to harden insecure services through the deployment of wrapper(s) designed to turn a possibly vulnerable service into a secure one. *HALE-IoT* will expose a secure interface of the service and will act as a proxy of the actual service while offering certain guarantees, such as confidentiality and secured access control. Central to this step is the retrofit of an SSL, or TLS, proxy that will: a) offer a cryptographic upgrade if the device lacks it, including the use of HTTPS instead of HTTP, SFTP instead of FTP, or SSH instead of Telnet and b) offer protection against SSL/TLS attacks (e.g., downgrade, MITM—Man In The Middle), and patch weak SSL/TLS configurations (e.g., hardcoded self-signed certificates).
2) *Proactive Attack Detection:* This steps aims to offer a proactive protection against application-layer attacks through the retrofit of a domain-specific firewall. For instance, *HALE-IoT* will implant a WAF when an IoT device processes Web HTTP connections either directly from the user through a Web browser or a RESTful client.
3) *Advanced-Level Access:* This step aims to harden a critical component of IoT devices, their admin interface. IoT devices do not generally have a graphical user interface, and their administration is generally done remotely.

The result of applying our methodology to hardening a generic IoT device is presented in Fig. 1. In this article, we offer an implementation of *HALE-IoT* that can scale the deployment of prehardened images for vulnerable legacy firmware that can benefit from a secure front end. We assume that these IoT devices expose services through the network while listening to a port through a socket. Our system performs best when there is a configuration file that specifies the network settings, and we restrict binary patching only to changes in the interface or the port number when these are hard-coded into the binary (Section VII-F). Note that more intrusive modifications are subject to less automation, thus making the solution less scalable and cost-effective. Also, more-intrusive modifications are highly likely to interfere with the normal intended operation of the given service, or even the entire device. While our methodology supports any type of binary patching, assessment of the impact they have on the fail-safe minimization principle is in the scope of our future work. We next describe in detail each of the three layers that constitute *HALE-IoT*.

B. Secure Front-End: SSL/TLS Hardening

An SSL/TLS Proxy is a specific type of proxy server designed to add a layer of SSL/TLS to protocols that lack this feature. For example, it is commonly used for adding HTTPS encryption to plain-text HTTP services without native HTTPS support. It is mainly responsible for the encryption and decryption of SSL traffic between the client and the server, and redirecting the packets, once decrypted, to the HTTP Web server. As mentioned, the rationale behind adding an SSL/TLS proxy is driven by the prevalence of IoT devices running insecure or weakly secured HTTP implementations [27], [28], [29] that give a *false sense of security*.

In our implementation, we used two approaches to SSL/TLS proxying: 1) SSL-proxy [41] as the main approach and 2) lighttpd [42] as an alternative approach. SSL-proxy is a project written in the Golang programming language. SSL-proxy features a high portability to other systems, making it a good candidate for systems that require multiple architectures. SSL-proxy allows self-signed certificates to be generated as well as working with existing certificates and full certificate chains, that are stored locally or generated through Let's Encrypt [43], [44]. For SSL-proxy cases, our toolsets generate Go binaries for the different architectures *HALE-IoT* supports, and then use the same SSL-proxy code in corresponding interpreted environments. As an alternative to SSL-proxy, we cross-compiled a statically linked version of lighttpd with SSL/TLS and proxy support, which, similarly to SSL-proxy, supports self-signed certificates or certificates generated through Let's Encrypt. In practice, it was only necessary to use lighttpd-based TLS proxy for the real device presented in Section VI-D; however, this lighttpd-based setup was also successfully tested on several other devices. For the purpose of our experiments, we used self-signed certificates, but we later discuss deployment issues in Section VII-G. However, the main idea of adding SSL-proxy is to provide any IoT device a guaranteed and uniform HTTPS support (e.g., latest TLS protocols) that can also operate proper full certificate chains [27], [28], [29].

C. Proactive Detection: Application-Layer Firewall Hardening

A generic application-layer firewall (xAF) is a type of firewall that can potentially detect and prevent malicious inputs

designed to exploit specific application protocols. Our architecture allows the retrofit of multiple xAFs, one for every potentially vulnerable network service. Therefore, *HALE-IoT* can both isolate local networks (from 0.0.0.0 to 127.0.0.1), and harden traditionally vulnerable services, such as Telnet (secured with SSH), FTP (secured with SFTP), UPnP, and MQTT. This architectural vision is presented in Fig. 1.

Our current implementation of *HALE-IoT* methodology primarily offers support for hardening Web services at the application level. A Web application firewall (WAF) is an additional security layer that inspects Web requests before redirecting them to their destination, allowing it to detect potentially malicious requests and avoid redirecting them to the Web server or to the Web application. When a malicious request is detected, the WAF is supposed to prevent the request from reaching the Web server, being able to detect the most common attacks at the Web application level, such as SQLi, remote command execution injections, XSS, or cross-site request forgery (CSRF) attacks. In particular, we use *Raptor* [30], which is a lightweight open-source WAF written in the C programming language. It has very few dependencies, making it a good candidate for use in embedded systems. Raptor adds an additional security layer that protects Web applications by comparing the content of HTTP requests with common signatures using a deterministic finite automata (DFA) algorithm. Additionally, its functionality can be extended with rules and other matching strings algorithms, such as Karpe Rabin, Boyer Moore Horspool, or perl-compatible regular expressions (PCRE). We cross-compile *Raptor* for the MIPSeb, MIPSel, ARMel, and Intel 80386 architectures, which are the ones currently supported by *HALE-IoT*. However, there is virtually no limitation to which CPU platforms *Raptor* (or any other WAF) can be cross-compiled for. At the same time, *HALE-IoT* could implant any other WAF as long as it can be either cross-compiled to native binary format for a device's CPU, or can run in a cross-compiled runtime environment (e.g., Python and Go). The only unavoidable limitation our system inherits stems from the constraints of the actual devices in terms of obsolescence of runtime, RAM memory, and flash storage (Section VII-E and VII-H).

D. Administration of HALE-IoT

HALE-IoT is composed of third-party components (e.g., WAF, xAF, and HTTPS proxy) that may require bug-fixes, improvements, and configuration updates over time. For example, there is also a constant evolution of the threat landscape (e.g., applicable vulnerabilities, working exploits) against which *HALE-IoT* offers protection to (legacy) IoT devices, and as such requires updates to the rules-sets of WAF and xAF. These and similar related factors dictate the need for a way to administer *HALE-IoT* in an easy, secure, universal, and low-footprint manner. A classical way would be to use a Web-interface to administer *HALE-IoT*, but here we opt for an SSH-based administration.

There are several reasons why we chose the use of an SSH-based interface for *HALE-IoT* administration instead of, for example, a Web-based administration interface. First, SSH by default has a proven and strong built-in authentication and authorization mechanism and protocol based on public-private key infrastructure (PKI). In the case of Web servers, it would require adding HTTP and/or HTML authentication models, which would add to the complexity of implementation and maintenance as well as potentially expose its own set of authentication/authorization vulnerabilities. Second, compared to a Web-interface, SSH does not require additional third-party dependencies and interpreters (e.g., PHP, Python) to provide full-fledged server-side functionality. With an SSH-based approach, the overall "application attacks surface" scheme remains generally the same even after adding the new SSH dependency. The Web-interface option, on the other hand, would increase the attack surface through addition of the Web server and the admin Web pages themselves. Third, SSH provides a simple yet powerful interface for performing additional system-administration tasks should the need arise (e.g., reboot, power-off, filesystem access). In the case of a Web-interface, there may be certain limitations to the administrative actions that would be available to the Web server or the Web pages. Last, but not least, efficient SSH implementations can be statically built with much lower footprint and overhead (e.g., Dropbear SSH at 100–200 kB). Such footprints are considerably lower than most Web servers coupled with runtime interpreters (e.g., lighttpd + PHP).

E. Other Types of Hardening

The simplicity and flexibility of the *HALE-IoT* approach is one of its core design principles (as stated in Section III-A), which is one of its strengths compared to the current state-of-the-art. An additional improvement by *HALE-IoT* would be the addition of hardening at networking layers L3–4 and L7. In essence, it would mean protecting all the interfaces and all the services in a generic whole-system manner against network layer attacks (L3–4), as well as against application layer brute-force attacks (L7). For layers L3–4, *HALE-IoT*'s architecture can integrate industry-standard tools like iptables, Snort, Suricata, Bro, and fail2ban for layer L7.

There are several adoption challenges that need to be considered. First, some IoT devices may not expose direct or standard access to various interfaces, thus requiring more intrusive reconfiguration, binary patching, or OS/kernel "hacks." Second, since some IoT devices may use less common OS flavors (i.e., other than Linux-derivatives), rebinding and configuration of network interfaces may be different and may require certain *HALE-IoT* implementation adaptations. We thus leave the implementation and evaluation of additional L3–L4 and L7 retrofits as immediate future work.

IV. Experimental Setup

To evaluate the effectiveness and efficiency of the *HALE-IoT* method, we applied it to Web services of a wide-range of IoT devices. We chose to harden and evaluate Web services as the immediate focus, because these are the most commonly present services on most IoT devices. For this, we retrofitted and emulated 395 firmware packages with defensive implants containing a WAF (for prevention of Web-attack

TABLE I
OUR INITIAL FIRMWARE DATA SET (BY VENDOR AND ARCHITECTURE)

Vendor	ARMel	MIPSeb	MIPSel	Intel/386	Total	*HALE-IoT*
Asus	0	0	1	0	1	0
Belkin	4	10	30	0	44	0
Buffalo	1	0	2	0	3	0
D-Link	76	54	1	0	131	37
Huawei	0	0	1	0	1	0
Linksys	2	3	19	0	24	8
Netgear	45	174	150	0	369	46
OpenWrt	1	146	166	7	320	293
Tenvis	0	0	6	0	6	0
Tomato by Shibby	8	0	171	0	179	0
TP-Link	2	152	5	0	159	0
Trendnet	3	25	9	0	37	11
Ubiquiti	8	3	0	0	11	0
Total	**150**	**567**	**561**	**7**	**1,285**	**395**

TABLE II
DISTRIBUTION OF THE INITIAL FIRMWARE DATA SET (BY WEB SERVER AND CONFIGURATION FILES)

Web server	# of FWs	# of FWs (config file)	# of FWs (*HALE-IoT*)
lighttpd	58	52	42
httpd	649	74	68
minihttpd	37	0	0
AppDemo	71	0	0
boa	46	44	7
uhttpd	390	314	278
webs	2	0	0
goahead	9	0	0
Not found	23	0	0
Total	**1,285**	**484**	**395**

vectors), and an HTTPS-proxy (for proper end-to-end HTTPS support).

In order to implant *HALE-IoT*, we identified the Web server configuration files and reconfigured them for hardening as follows. First, taking into consideration the firmware's CPU architecture, we copy corresponding cross-compiled files to the firmware filesystem as the implants needed by *HALE-IoT*. This includes executable and other files for the hardening elements (Raptor, SSL-proxy, and Dropbear), configuration and rules-set files, and authentication keys for the *HALE-IoT* SSH sysadmin interface. Then, we add the initialization scripts of the tools to the set of scripts that will be executed once the booting process finishes (e.g., init.d, rc.d, and registration.d). Finally, we reconfigure the Web server configuration files or Web server initialization scripts to isolate the interface and listening port of the service (e.g., original Web server rebind to 127.0.0.1:81), and then we start full-system firmware emulation [2], [3]. This process was *fully automated*, and was carried out for each test of the evaluation.

A. Data Set

Our initial data set consists of 4809 real-world firmware images extracted from FIRMADYNE [3]. Note that the original FIRMADYNE data set is larger, but 4809 images are available to download at the time of writing. We then retain only the images in the architectures that *HALE-IoT* currently supports (i.e., ARMel, MIPSel, MIPSeb, and Intel 80386, cf. Section III), making a total of 1328. From these, we discard 43 images that have a custom format compression algorithm and, thus, cannot be systematically unpacked with Binwalk [45] (which comes as part of the FIRMADYNE setup). After processing all remaining images, we managed to extract the root filesystem from 13 device vendors (ranging from Asus to Ubiquity) for 1285 images overall. Overall, these firmware root filesystems are associated with devices of the following type: Ethernet routers, WiFi routers, xDSL modems, and IP cameras. Table I shows the distribution of vendors in our data set, per CPU architecture.

It is important to note that when trying to address such an immense and heterogeneous experimental population and space, for practical and resource reasons we are bound within magnitudes that are feasible for handling such experiments. At the same time, our work exceeds comparable experimental state-of-the-art works such as Firmalice [46] (data set size: $N = 3$ samples), and generally positions our experiments within the magnitude range of similar recent works, such as FIRMADYNE [3] and Costin et al. [2] (data set size: $N = K \times 10^2$, i.e., hundreds of samples).

B. Emulation

To evaluate *HALE-IoT*, we emulate a device that runs the firmware images in our data set. Since we are mainly interested in systems that have a Web interface to administrate the device, we next describe the steps we take to select those images. We first scrape the file system of the image to look for binaries that are core components of a Web server (e.g., uhttpd). We then identify the configuration files that inform settings to the Web server (e.g., boa.conf and lighttpd.conf). Table II shows the different types of Web servers together with the number of firmware images (marked as "FWs"). As expected, a large subset of images have a Web server configuration file together with the server binary, an exception being uhttpd. Images with a binary and without a configuration file have the server settings embedded in the binary itself. To scale our evaluation, we focus primarily on the 484 images that have an explicit and nonembedded configuration file.

We note that from architecture and design perspectives, *HALE-IoT* can run virtually on any type of firmware as long as the user(s) can change the binding network interface and port of the service that we aim to harden. However, in certain cases (e.g., service uses custom or binary-hardcoded configuration), changing the network interface and the port may require more manual effort, and we discuss such challenges in Section VII. In the end, out of all 484 images that have a Web server configuration file, we managed to successfully emulate and implant *HALE-IoT* to 395 firmware images. The emulation and *HALE-IoT* implant covers the following five vendors: D-Link, Netgear, TRENDnet, Linksys, and OpenWrt (Table I) and the following four Web servers: lighttpd, httpd,[4] boa, and uhttpd (Table II). Once the emulation started, we were able to successfully communicate with all 395 Web server processes and, more importantly, we were able to retrofit the *HALE-IoT* security hardening measures in all of these firmware images.

[4]In most IoT devices we encounter, "httpd" is just a generic placeholder name for the Web-server, and should not be assumed to be Apache's HTTP server.

C. Toolsets

One key aspect of *HALE-IoT* is that it supports different out-of-the-box CPU architectures and is flexible enough to keep adding more architectures and defenses in the future. In particular, we compile our framework for ARMel, MIPSel, MIPSeb, and Intel 80386, as previously discussed. While it is possible to use QEMU to emulate each operating system used by the different vendors, we opt to perform a systematic cross-compilation through a toolchain. There are different toolchains available, including Linaro [47], or Linux MIPS Toolchain [48]. For the purpose of this article, we created our own customized toolchain using *Buildroot* [49]. Our toolchain uses *musl* [50], which implements the standard C library with some improvements, such as enhanced support for static linking. When cross-compiling the different binaries using our toolchain, we strip the binary of all symbols to optimize size. At present, we automatically cross-compile *Raptor* (for WAF), and *Dropbear* (for *HALE-IoT* SSH-based administration) for all the supported architectures. We do not cross-compile SSL-proxy, as it is written in the Go programming language and the binaries for the different architectures can be generated directly without using a specific toolchain. Final builds of the toolsets resulted in the following footprints: Raptor 275.8–346.6 kB, SSL-proxy 5053.5–6244.3 kB, Dropbear SSH 179.8–228 kB, and lighttpd 2381–3018 kB. For additional resource overheads incurred from *HALE-IoT* implant, see Section V-E.

It is important to note that our experimental setup is systematic and easily extensible to other architectures and defensive toolsets, which is precisely the scope of our future work, as discussed in Section IX.

V. Test Methodology and Results

A. Test Methodology

We run two tests for each QEMU-emulated firmware following a DevOps methodology [51], [52]. This methodology evaluates changes into a system in an incremental fashion so that failure causality can be properly attributed. One test contains a hundred common (i.e., nonexploiting) Web requests, while the other test has a hundred Web requests with some type of Web-attack payload (e.g., XSS, SQLi, CI).

Our DevOps-style testing methodology has the following steps. First, we emulate the firmware without any kind of modification and run the tests as a control measure to evaluate the differences. We also check how many firmware images accept connections through the HTTPS. Then, we retrofit the firmware with the Raptor WAF and launch both tests again. We do the same again but only after implanting the SQL-proxy in the firmware. Finally, we launch both tests on the firmware emulated with both protection measures retrofitted, that is, the Raptor WAF and the SQL-proxy working together. We also ensured that random nonmalicious requests return exactly the same result in both tests (i.e., with and without *HALE-IoT*). By comparing the HTTP headers and the content returned in both test setups (normal versus nonmalicious) requests. To assess the performance of the devices after retrofitting the WAF in a realistic setting, we use Raptor's DFA algorithm and 55 regular PCREs we gathered from the community [53], [54].

TABLE III
Correctness and Effectiveness of the Retrofit 395 Emulated Web Servers

Test under evaluation	# of emulated FWs	(%)
Firmware is *functional* after retrofit	395	(100%)
Isolation *OK* (web service re-bind to 127.0.0.1)	363	(91.9%)
Isolation FAIL (web service hardcoded to 0.0.0.0)	32	(8.1%)
Default HTTPS present before retrofit	84	(21.3%)
Default HTTPS *missing* before retrofit	311	(78.7%)
HTTPS present *after retrofit*	395	(100%)

We apply our test methodology to answer the following questions.

- Q1) Is *HALE-IoT* able to retrofit defensive firmware modifications and implants without disrupting the normal operation intended for the firmware?
- Q2) Can *HALE-IoT* effectively deploy a secure front-end in legacy devices?
- Q3) Can *HALE-IoT* effectively deter known attacks and known vulnerabilities against legacy devices?
- Q4) Can *HALE-IoT* effectively cover multidimensional heterogeneity (e.g., physical versus emulation, ARM versus MIPS versus x86, real CVEs versus synthetic vectors, open-source versus proprietary, cross-vendor)?
- Q5) What is the potential performance overhead incurred by *HALE-IoT*?
- Q6) Can *HALE-IoT* actually work with services that apparently cannot run on 127.0.0.1:<*port*> via configuration file?
- Q7) Finally, does *HALE-IoT* actually work on physical devices, rather than just emulated environments?

We evaluate Q5 using emulation only, Q7 using bare-metal hardware only, and Q1–Q4 and Q6 using both bare-metal and emulation.

B. Correctness and Effectiveness of the Retrofit (Q1)

To test how well *HALE-IoT* retrofits defensive firmware modifications, we deploy all hardened images in our own emulator environment resembling the one in [2] and borrowing additions from `FIRMADYNE` (e.g., NVRAM) [3]. Table III shows a summary of our results. We see that all 395 images remain functional, i.e., the hardening process does not disrupt the normal operation intended for the firmware. However, some cases underperform in terms of isolation. In particular, we see that in 8% of the images we continue to see the Web server listening in the external interface (0.0.0.0), and, thus, potentially exploitable connections to the original Web server are possible without going through our hardening proxy chain. We also see that 78% of the emulated firmwares do not use HTTPS by default before the retrofit. We next explore in detail the performance of *HALE-IoT* when looking at the first layer of its architecture (cf. Section III-B).

> During our *HALE-IoT* experiments, neither our human experts nor our automated tools have encountered any functional abnormality, and the emulated-and-hardened Web services, along with the entire

system emulation, performed normally and as expected.

C. Secure Front-End in Legacy Devices (Q2)

One of the main hardening goals of *HALE-IoT* is to isolate the vulnerable services from attacker-accessible interfaces (e.g., WAN, LAN), while at the same time keeping the original services running on 127.0.0.1 to satisfy Q1. Our evaluation shows that *HALE-IoT* successfully reconfigures original Web-servers from 0.0.0.0:80 into 127.0.0.1:81, replacing the former address with the service running our WAF implant while relaying only safe HTTP Web requests to the original Web server now residing in the latter address.[5]

Further analyzing the results shown in Table III, we make two key observations. First, 100% of the original Web-servers (from the successfully emulated 395) rebind well to port 81 as instructed by *HALE-IoT*'s reconfiguration routines. Second, despite being explicitly instructed to change binding from 0.0.0.0 to 127.0.0.1, there are 32 firmware images that remain bound to 0.0.0.0 (in addition to the new address). This can expose a potentially vulnerable service to attacker-accessible interfaces, thus rendering our hardening ineffective. We posit that this is due to vendors' (un)intentional implementation and coding choices or errors, where only some values from the configuration file(s) are taken into consideration while the rest of the parameters are either hardcoded into the binary executable or taken from other nonobvious configuration files. We evaluate *HALE-IoT* for this case in Section VI-A, and discuss this challenge in more detail in Section VII, but we emphasize that this happens in only 8% of our images.

Another of our aims is to use SSL-proxy to add secure tunnel wrappers around services. Our rationale is that these services either have weak secure tunnels, or are just plain-text altogether (i.e., adding HTTPS support to IoT devices that quite commonly lack it). As discussed, in certain instances, the IoT device may provide an HTTPS server by default. In 84 emulated firmware images, the original Web servers also start a "default HTTPS server." However, besides carrying a self-signed certificate, the default HTTPS server also featured an outdated SSL/TLS version (e.g., TLSv1), hence very likely exposing the Web interface to various HTTPS and MITM attacks. On the other hand, with *HALE-IoT* (e.g., with SSL-proxy) we are able to provide the hardened IoT devices with the latest and most secure TLS implementations, along with the proper support for full certificate chains (see also discussion in Section VII-G). This, in turn, provides real increased security rather than merely a "sense of security" provided by most *default HTTPS servers* when these are implemented in IoT devices and working with self-signed or expired certificates.

Overall, *HALE-IoT* automatically manages to fully isolate 92% of the potentially vulnerable Web services, while correctly providing a secured SSL tunnel in 100% of the tested cases.

[5]When WAF is chained with SSL-proxy, the WAF is further isolated to 127.0.0.1:80, and Web service is exposed by SSL-proxy binding to 0.0.0.0:443.

D. Detection and Prevention of Attacks and Exploits (Q1, Q3)

To evaluate the performance of *HALE-IoT* in regard to its second architectural layer (Section III-C), we perform two experiments.

Automated-Attacks: We leverage a battery of 200 Web requests, of which half are common requests and the other half are known Web attacks. The attacks include known XSS, SQLi, and CI attacks coming from both actionable CVEs and synthetic input. Our results show that *HALE-IoT* can detect all known attacks when configuring the vanilla WAF community detection rules. The detection rate itself is not at all surprising, but this experiment reports a valuable finding: *HALE-IoT* can reliably retrofit complex defense mechanisms into the firmware of IoT devices through binary retrofits while keeping the original firmware functional (Q1), and offering the full-fledged level of protection of the retrofitted secure mechanism (Q3).

Targeted-Evaluation: We also evaluate the effectiveness of *HALE-IoT* by targeting some firmware images with CVE-2016-1555 (also known as ACSA-2015-001). The CVE-2016-1555 was independently discovered by Chen et al. [3] and Costin et al. [2]. This known vulnerability covers a series of preauthentication XSS and RCE in several devices from Netgear (many of which are already EOL, and therefore will remain unprotected indefinitely unless it gets hardened with *HALE-IoT* or similar). First, we exploit the vulnerabilities in the emulated environment and confirm that the original firmware is vulnerable and exploitable. Then, we apply *HALE-IoT* to the emulated firmware and see that all attacks are efficiently stopped. This further proves the effectiveness of our approach, but this time with an attack that targets EOL devices. We refer the reader to Appendix-A for visual representation of the success of our proof-of-concept attack and defense.

Takeaway: HALE-IoT can effectively deter known attacks against legacy devices. Naturally, our system inherits the limitations of the defense mechanism we implant. In particular, Raptor is mainly effective at detecting known attacks and can miss connections that include zero-day Web attacks. We discuss this limitation in detail in Section VII through different axes, including the WAF's inherent limitations (Section VII), and limitations in the data sets (Section VII-C). However, we also note that the overall effectiveness of *HALE-IoT* when it comes to the detection of attacks has to be seen from an holistic perspective. In Section V-C, we report the effectiveness of our system at hardening insecure (superfluous) services other than HTTP. When putting together the secure front-end and the proactive detection layers (Fig. 1), *HALE-IoT* can offer a system resilience to both known attacks against Web services, and against unknown attacks targeting all other hardened services.

With *HALE-IoT* implanted, we achieve a 100% detection and prevention rate of *known attacks* in both emulated and real-devices, while effectively hardening other services that are often target of unknown (zero-day) attacks. This 100% detection ratio is taken as a unit test rather than a detection ratio. This provides assurances that the WAF we retrofit

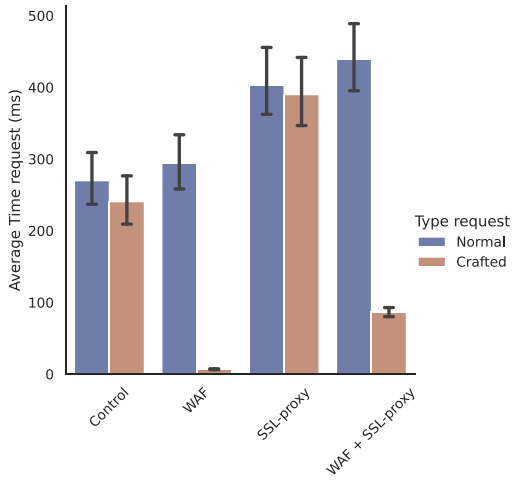

Fig. 2. Average time to complete the HTTP request for each test run.

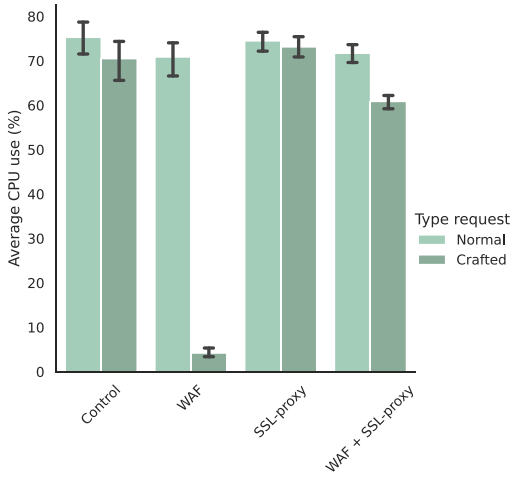

Fig. 3. Average CPU consumption for each test run.

works as expected under active attacks. We are aware that WAF systems detect attacks for which there is a known rule, and they are unquestionably subject to evasion, just like any other rule-based detection system—whether the WAF is on a high-end production system, VPN appliance, or a *HALE-IoT* retrofitted router/camera. However, they provide an extra layer of security that protects against known exploits targeting the firmware's Web interface, and they prevent most automated attacks (i.e., via bots looking for vulnerable devices) that target vulnerabilities in the exposed Web servers of IoT devices [5], [35].

E. Functional, Performance, Overheads Evaluation (Q4, Q5)

We have collected measurements of the performance overheads introduced by various components of the *HALE-IoT* implant. Since *HALE-IoT* is highly flexible and configurable, we use a modular analysis to assess our performance. That is, we measure the performance of the WAF alone, the SSL-proxy alone, and the SSL-proxy chained with WAF. For each test we collect benchmarks for the CPU and memory consumption, as well as the response time of the Web requests. The performance evaluation in the emulation provides an approximation of the memory and CPU consumption as the difference between runs with and without any type of retrofitted tool.

In Figs. 2–4, the references to "WAF," "SSL-proxy," and "WAF + SSL-proxy" represent the use of *HALE-IoT* with a particular self-descriptive configuration, while "Control" represents firmware emulation without any added components. As we discussed in Section V, each test is made up of one hundred common Web requests represented as "Normal," and one hundred requests that contain some type of attack represented as "Crafted." We carry out all tests in each firmware that we emulate and implant *HALE-IoT* into. In total, our evaluation scripts made 316 000 Web requests. Figs. 2–4 represent the average of the results over the entire set of emulated and tested firmware images. The data was collected using common Linux tools (e.g., mpstat or vmstat) from the host side.

We also provide an interpretation of the performance overhead graphs. In Fig. 2, we see that the response time for Normal requests increases proportionally to the number of

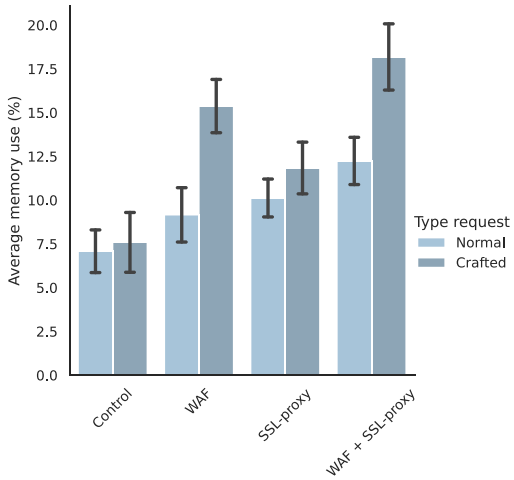

Fig. 4. Average RAM memory consumption for each test run.

chained components for the particular hardened service. For example, "WAF + SSL-proxy" request time takes longer than "WAF" or "SSL-proxy" separately. This is more or less expected, as in the case of WAF + SSL-proxy, the request is forwarded back and forth via multiple connections and software modules that have their own context-switching delays, etc. At the same time, Fig. 2 shows that in the case of "Crafted" requests, whenever the "WAF" component is present, the complete request time is significantly lower than Normal. This is both expected and direct evidence that the WAF effectively detects and blocks attack attempts, and as such protectively terminates HTTP communications carrying potentially malicious payloads at much earlier stages. A similar pattern can be seen in Fig. 3. In the case of Crafted requests, whenever the WAF component is present (e.g., when only WAF is present without SSL-proxy), the average CPU usage is lower than Normal. Once again, this is both expected and direct evidence that the WAF effectively detects and blocks attack attempts, as WAF does not continue any further computations and processing (e.g., relaying it to the original Web service) once it has detected and prevented potentially malicious payloads.

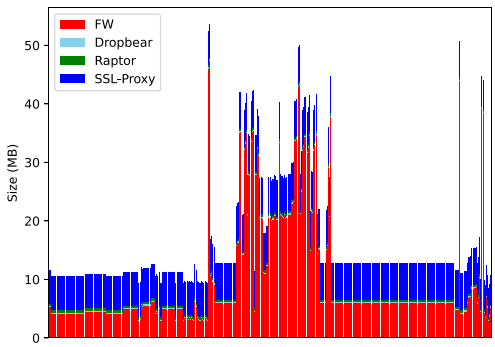

Fig. 5. Size of the FWs along with the size of the retrofitted binaries.

Moreover, the average memory consumption increase shown in Fig. 4 is also expected, as the additional components require memory for operation and for storing their data, such as WAF rules-sets and SSL/TLS certificate chains. However, the memory consumption is harder to fully interpret than CPU usage and processing time of requests, as the coding practices can vary greatly across the applications. Also, as opposed to CPU usage, which stops when a particular function flow stops (e.g., HTTP request blocked and terminated), the memory is often not immediately freed (or not made visibly available to OS, even if freed by the application) when the program reach certain states such as when a Crafted packet is detected and blocked. In terms of storage overhead introduced by *HALE-IoT*, Fig. 5 shows the distribution of sizes for all firmware images along with the retrofitted binaries. Specifically, as detailed in Section IV-C, the increase by each components is as follows: Raptor 275.8–346.6 kB, SSL-proxy 5053.5–6244.3 kB, and Dropbear SSH 179.8–228 kB.

During the evaluations presented in this work, we did not perform an *exhaustive regression testing* on the Web interface (nor on other functions and services) operating within the evaluated devices and emulated firmware. Because such exhaustive complete system regression testing would be a nontrivial experiment in itself, we leave as future work the large-scale evaluation of the functional impacts induced by retrofit defense systems such as *HALE-IoT*. However, we performed an evaluation of the retrofitted software. We apply fuzzing by creating a harness for the core component of *HALE-IoT* (i.e., the function to which any data received through the socket is passed) and use american fuzzy lop (AFL) as a fuzzing tool. We see that the routines added as part of *HALE-IoT* are safe and do not produce any crashes (100% success for all tests). However, we found some memory-related bugs and several crashes associated with an old and vulnerable version of the WAF we used in our initial experiments. We report this finding next to show the importance of performing black-box testing using fuzzers, but we note that our final implementation of *HALE-IoT* uses a WAF that was not vulnerable to this bug and did not report any crashes at the time we tested it with the fuzzer.

The crashes in the old version of the WAF occurred in *waf-mode* four (one of the command line options). This parameter has four levels of protection, number four being the highest, and defines the mode of the DFA algorithm to detect common attacks. With the rest of the modes, and with DFA disabled using only regular expressions as rules, the application did not produce any crashes with the same test cases. After further inspection and debugging, crash occurred when trying to read a value beyond the stack limit, which causes a segfault. This error is caused by the use-after-return memory error, and since this memory area belongs to a function that has already terminated, it can cause undesirable—yet not exploitable—behavior. WAFs are both research-wide and industry-wide accepted methods for securing Web-apps (including legacy Web-apps) [37]. We discuss the bugs found, as well as the security limitations that any retrofitted piece of software may place on a system in Section VII-I.

A differentiating end goal of *HALE-IoT* with respect to related work (e.g., ABSR and Symbiotes [55]) is that we aim to be as unintrusive as possible, and to ensure that legitimate requests do not have in important impact on the performance of the device. Our results strongly support this goal.

> We see that the use of *HALE-IoT* introduces some interesting tradeoffs. When attacks are blocked, we effectively reduce the overhead. Judging by the performance of the hardened device when processing legitimate requests alone, we see that *HALE-IoT* does not introduce an important overhead.

TABLE IV
SUMMARY OF REAL DEVICES PERFORMING SUCCESSFULLY IN OUR EVALUATION

Device	CPU / Cores / MHz	Architecture	RAM	Storage
Asus RT-N12+ B1	MT7628NN / 1 / 580	MIPSel	32 MB	8 MB
Netgear R6220	MT7621ST / 1 / 880	MIPSel	128 MB	128 MB
Linksys EA4500	88F6282 / 1 / 1200	ARM	128 MB	128 MB
RaspberryPi 3	BCM2837B0 / 4 / 1400	ARM	1024 MB	4096 MB

VI. CASE STUDIES

We next present a number of case studies that aim at a better understanding of the performance of *HALE-IoT* in detail. In particular, we look at firmwares image from *Linksys* and *Asus*. We conclude our case study with the deployment of a hardened version of *Asus RT-N12+ B1*, *RPi3 OpenWrt*, *Netgear R6220*, and *Linksys EA4500* over four different hardware devices. Table IV summarizes the technical specifications of the actual physical devices used in our evaluation.

A. Reverse-Engineered Hardcoded Binary for Linksys wrtsl54gs (Emulation) (Q6)

As presented in Section V-C, there were 32 emulated firmware that failed to isolate webserver via binding to 127.0.0.1. For unknown reasons, the firmware kept the Web service binding to 0.0.0.0. In order to demonstrate that *HALE-IoT* is also feasible, practical, and effective even when the reconfiguration retrofitting fails, we attempted a minimal-effort manual reverse-engineering of one such Web-server binary.

For this, we chose the *httpd BusyBox* Web server binary from OpenWrt firmware built for wrtsl54gs device by Linksys. Even though *httpd BusyBox* is known to support the "$-p$" option to change the binding interface and port (e.g., "$-p$ 127.0.0.1:81"), in this particular case it was not supported or

it did not work. We then investigated the potential reasons behind this failure. The wrtsl54gs firmware image has a non-stripped BusyBox binary that is dynamically linked, so our first approach was to look for HTTP functions to recognize the httpd BusyBox applet. Then, we identified the call to the `bind` function and checked the parameters backward. We found the `inet_aton` function that converts a string IP address into binary form and that uses the variable assigned from the "$-l$" command-line argument as a parameter. Though this argument does not appear in the help menu of the httpd command, it allows the listening interface of that specific httpd binary to be changed. We leverage the hidden *-l* option to successfully run *HALE-IoT* in the wrtsl54gs firmware.

We can further generalize this *one-time manual effort* into *HALE-IoT*'s automation as follows. We can identify similar service-exposing binaries using, for example, Yara rules [56], or heuristics and matching based on the op-code level or semantic code-similarity [57]. Similar binaries could relate to: 1) the same device (but different firmware version); 2) the similar device models (from the same vendor); or even 3) distinct devices across vendors (e.g., "white label" products). The takeaway from this case study is that manual efforts can sometimes provide "intelligence" that can help to scale the hardening of images over a very large number of similar firmware environments.

B. Evaluation on RaspberryPi With OpenWrt (Device) (Q7)

To evaluate *HALE-IoT* over a bare-metal device, we deploy an OpenWrt (LEDE 2017 build) into a RaspberryPi 3 device. OpenWrt is the most popular vendor in our data set that has firmware images for all our architectures. The LEDE 2017 build version of OpenWrt has a known XSS in its LuCI Web interface.[6] Therefore, we first run a nonhardened OpenWrt firmware and we see that the vulnerability can be exploited in practice (see Fig. 8 in Appendix-A). We then harden the same OpenWrt firmware with *HALE-IoT* and see that with the WAF + SSL-proxy configuration we can completely prevent the XSS attack, in addition to being able to add full HTTPS support (see Fig. 9 in Appendix-A). This case study indicates that *HALE-IoT* works as expected on bare-metal devices. Our next case study explores this further.

C. Evaluation on Asus RT-N12+ B1 (Device) (Q6, Q7)

We evaluate *HALE-IoT* on another bare-metal device we had access to; Asus RT-N12+ B1. This device runs MIPS32 binaries, and, in particular, uses a custom httpd as its Web server. This image requires a retrofit at the binary level, as the configuration parameters of the Web server are hardcoded into the binary and can not be identified by *HALE-IoT* automatically. After reversing it, we see that the binary accepts three arguments: 1) the name of the interface whose IP address will be obtained through `SIOCGIFADDR` ioctl; 2) the port; and 3) a way to enable SSL connections. The device also runs and exposes Telnet and SSH services that we used for "live implanting" (Section VII-A).

As a result of our implant, we see how *HALE-IoT* spawns the WAF into the device, and how the Web service is secured behind HTTPS while hardening all other services. We see that Raptor works as expected, detecting and preventing potentially malicious input test-vectors. However, we notice that the device periodically faced some resource limitations manifested as unavailability of RAM memory. Nondeterministically, when insufficient RAM is available for handling HTTPS/HTTP/network requests via the *HALE-IoT* processing chain, the spawned process/thread (e.g., WAF, SSL-proxy) is killed by the OS/Kernel due to lack of sufficient memory blocks to allocate. This is a limitation rooted in a combination of technical factors such as the hardware runtime environment (i.e., device with very limited RAM), and the implementation choices (i.e., SSL-proxy executable size). However, this case study shows how our methodology can harden Asus RT-N12+ B1. In practice, for this type of device, a more lightweight defensive mechanism would have to be deployed in order to make the added defenses effective and usable. We further discuss the implications that drive the choices of the implants in Section VII-H. This case study shows that our generic methodology lets us work with heterogeneous firmware images, and also on bare-metal devices and firmware.

D. Evaluation on Netgear R6220 (Device) (Q6, Q7)

Next, we evaluate *HALE-IoT* on Netgear R6220. This device runs MIPS32 little-endian binaries, and uses the *mini_httpd* Web server. The Web server does not contain any configuration files, and the server options are configured through the command line. Therefore, we can change the listening port and interface via the server's arguments. We retrofitted Raptor and SSL-proxy on the device through a Telnet server that can be enabled in debug mode. Raptor worked as expected, but SSL-proxy did not work due to Golang incompatibilities with older MIPS kernel versions.[7] As an alternative to SSL-proxy, we use a statically linked cross-compiled lighttpd server with support for SSL/TLS and reverse proxy. We use a configuration that listens on port 443 with SSL enabled and redirects incoming requests to the WAF. As a result of using lighttpd as an alternative, we can see that the extra security layers added by *HALE-IoT* are working correctly. This case study shows us that our methodology is functional, flexible, and independent of the type of tools used.

E. Evaluation on Linksys EA4500 (Device) (Q6, Q7)

Finally, we evaluate *HALE-IoT* on the Linksys EA4500 device. This device runs ARMv5 and lighttpd binaries as a Web server. The default firmware does not present any access to the command line, which poses a challenge to *HALE-IoT*. However, we found a workaround that shows how our system can be deployed through unconventional means. Linksys EA4500 allows a user to connect universal serial bus (USB) devices to the router to share files over the network. When a USB is plugged in, it is mounted in the `/tmp` folder

[6]More details here: https://github.com/openwrt/luci/issues/1731.

[7]https://github.com/golang/go/wiki/MinimumRequirements

of the device. If a folder named `packages` exists, it is symlinked directly to the `/opt` directory. Finally, whatever file is present in `/opt/etc/registration.d/`, it will be executed by the shell.[8] Therefore, we use this hack/vulnerability to add a statically linked version of dropbear and *HALE-IoT* into the device's *running firmware*. As a result of this implant, Raptor WAF and SSL-proxy work properly together with access to the device via SSH to update or modify its configuration.

VII. Challenges and Discussions

A. Delivery of the Retrofitted Implants

Modifying an existing firmware is the first step in the delivery of an implant, and it can be done by leveraging tools like firmware-mod-kit (FMK) [58]. However, in certain cases, implants are not easy to realize in practice. This happens, for instance, when the firmware update needs a digital signature, or there is a cryptographic protection (e.g., strong and secured private key, correct implementation of validation). However, there are also vulnerabilities that allow flashing a noncertified or modified firmware into a device with these restrictions. Some of these vulnerabilities relate to forging digital signatures or bypassing digital signature verification. Giese [59] exploits a domain name system (DNS) redirect to trick Xiaomi Cloud to download modified firmware from a local server. Another example is when there is no firmware update available, except the original firmware running on the device. Finally, low-level frameworks like Firmware-Mod-Kit may be unable to support the specific firmware format that requires hardening. We next discuss alternative methods that *HALE-IoT* could deploy to circumvent this limitation. These methods revolve around the idea of making the implant directly into the device in runtime.

The first option is the use of network or serial interfaces (e.g., Joint Test Action Group: JTAG, Universal Asynchronous Receiver-Transmitter: UART) to access the built-in Telnet and SSH services via the bootloader or the OS prompt. Then, we can implant *HALE-IoT* using automation scripts over traditional OS sysadmin techniques, as shown in the case study in Section VI-C. The second option is to exploit a known vulnerability in the running device, such as remote code execution (RCE) or command injection (CI), to inject benign code and implant the *HALE-IoT*, for example, as demonstrated for Linksys EA4500 (Section VI-E). Naturally, *HALE-IoT* can then also patch those particular vulnerabilities so that they cannot be further abused. Note that similar techniques have been used by both highly competitive malicious botnets and vigilante IoT malware [60]. While this section discusses the challenges of modifying firmware (software), we next look at the issues behind dealing with the actual devices (hardware).

B. Persistence of the Retrofitting Implants

The IoT realm is heterogeneous, and the process of retrofitting additional security into these devices is a highly technical task. One task that remains particularly challenging is keeping these retrofits persistent across reboots and power-offs. *HALE-IoT* is stored at the filesystem level (e.g., flash storage) to maintain persistence. However, several factors can prevent *HALE-IoT* from being persistent, including factory resets, firmware upgrades, forceful flash storage cleanup (e.g., SPI communication with flash chipset), or even protections from manufacturers (e.g., restricting partitions to *read-only*). In many cases, the implants (both benign and malicious) can survive such "cleanup" scenarios by installing an implant component at the bootloader level, thus essentially acting as a boot-time rootkit. However, this is a challenging research area that requires further explorations and ethical considerations. Other specific protections from manufacturers can be overcome with case-by-case-basis techniques, for example, the restriction of partitions to *read-only* could be overcome by repacking or reflashing the firmware (Section VII-A).

C. Data Set Size and Representativeness

In order to analyze, harden, and test our *HALE-IoT system*, a firmware data set is required. The vendor's website is the first choice for gathering firmware, but third-party websites also host firmware images. The most convenient way to acquire firmware online is via Web-crawlers [1], [3]. However, harvesting a data set through Web-crawling has its limitations. For example, firmware that was once available online is often pulled offline by the vendor. This can threaten the reproduciblility of the evaluation. For example, state-of-the-art projects, such as `FIRMADYNE` [3], face this problem and we see a gap from the time of their release to the time of our experiments, such that many URLs and firmware are not available online anymore [1], thus limiting the experimental data sets from the start. Even if the crawlers can be updated to work with a redesigned website (which is tedious and does not scale in effort), they cannot fundamentally be fixed to download a firmware file taken offline by the vendor.

Also, some devices do not have firmware images available online. This could be due to the nature of the product or the age of the device. Pulling the firmware out of a device through Telnet or secure shell (SSH) connections is possible in some scenarios [61]. However, in many cases, memory dumps through hardware hacking is the only viable option [39], [62]. IoT devices often accommodate low-level hardware interfaces, such as UART or JTAG, through which it is possible to connect directly to the device's bootloader or to its root shell [39], and then take a storage and memory snapshot, or just perform a live-implant. This approach requires extensive human expertise and interventions, and also does not scale well.

A fundamental challenge to all research targeting IoT devices and firmware is the lack of a highly representative baseline of IoT firmware data set. Building a data set is challenging and tricky from multiple perspectives. On the one hand, collected firmware can face copyright scrutiny from vendors if it includes proprietary firmware. Also, it is highly unlikely that many relevant and omnipresent vendors will sign-off releases of their firmware into such a data set. On the other hand, ignoring proprietary images and including only

[8]Dan Walters: https://web.archive.org/web/20120914060622/http://blog.danwalters.net/.

open-source firmware would be easier to accomplish, but this would bias the data set and make it unrepresentative of the myriad of COTS devices running proprietary firmware. Our evaluation uses the FIRMADYNE data set [3], which is considered state-of-the-art. We were able to successfully process 395 images from it, which is comparable in size to the data sets used in prior works [2]. However, given the large number of IoT vendors, this data set can be seen as limited. Future work is needed to create a data set that is:

1) *a highly representative* baseline of IoT firmwares (i.e., multidimensional representativeness: CPU architecture, OS, device type, core services and functionality, networking interfaces and stacks, and firmware packaging formats);
2) *not facing licensing issues* (i.e., firmware that is proprietary, nondistributable, etc.);
3) *stable and always available* for download, duplication, and improvement (i.e., never lost, either partially or totally).

D. Firmware Obfuscation

Firmware images are often packaged and compiled, thus, requiring preparation before analysis [1]. Specialized software, such as Binwalk [45] and binary analysis-NG (BANG) [63], is used to unpack the firmware, revealing the file system and other information, thus enabling further analysis. However, as there is no standardization, some manufacturers try to obfuscate and complicate unpacking and reversing their firmware, for example, by adding custom format compression [1]. Due to memory and other resource constraints, IoT devices often ship with file systems designed for constrained devices, such as squash file system (SquashFS) or Journaling Flash File System (JFFS, JFFS2) [64]. These file systems are often read-only and have file system compression enabled. Additionally, software such as Firmware-Mod-Kit (FMK) [58] is one of the few available and one of the most popular tools to perform firmware modifications on a relatively wide range of formats and devices.

When not performing a live implant, *HALE-IoT* focuses on retrofitting legacy binary firmware via firmware modifications. Thus, it requires and performs: 1) the unpacking, and modification steps (if emulation is involved) and 2) the unpacking, modification, and also *repacking* (if a physical device is required to force a firmware upgrade for the implant to work). In these cases (especially when a physical device with a firmware upgrade is involved), in the end must produce a firmware that is accepted by the device and is fully functional. However, even though both firmware unpacking as well as modification-and-repacking are represented by existing toolsets to some extent, the current state-of-the-art does not addresses the fundamental challenges of unpacking and modification-and-repacking of nontrivially obfuscated or encrypted/signed firmware. In this sense, the *HALE-IoT* system inherits all the limitation of the existing tools (e.g., Binwalk, BANG, and FMK), which however is not a limitations of the *HALE-IoT* methodology itself. In our current evaluation, the physical devices and the emulated firmwares were representative of the IoT device populations that allow relatively easy live-implanting as well as unpacking, modification, and repacking. We posit that more work is required to overcome the analysis of obfuscated or encrypted firmware packages.

E. Runtime Environments Being Obsolete

Runtime challenges became obvious when we started experimenting with pushing implants into random COTS IoT devices for the purpose of our case study. We next discuss some of those challenges to illustrate the complexity of the problem space and elicit research efforts toward better instrumenting obsolete runtime environments.

In one instance, the router undergoing hardening had the vendor's original firmware, and was running BusyBox, Linux kernel, and other executable files compiled for MIPS-I. However, the Buildroot environment we use (including many of its prior versions), while producing MIPS32 builds, does not produce MIPS-I cross-compilations anymore.[9] For example, even though we tried to run on the router our toolsets precompiled by Buildroot for MIPS target, certain binaries returned errors such as *Illegal instruction.* This is the most tangible confirmation of a mismatch between the hardware CPU instruction set architecture (ISA) and the ISA generated into the executable by the cross-compilation.

Addressing the "obsolete firmware environment" challenge is important for several reasons. Any system, whether offensive [55] or defensive such as *HALE-IoT*, will most likely face either exactly the same or very similar challenge on a constant and increasing basis. Some reasons for this are that devices become obsolete/EOL faster, and the technology and software development life-cycle is constantly accelerating. The above, in turn, implies several more things. First, supporting many legacy IoT devices will require an ever growing toolbox of cross-compilation environments. Such a backward compatibility toolbox should provide as complete coverage as possible in terms of combinations for CPU ISA, OS/kernel, application binary interface (ABI), and runtime environments (including all different versions and intercompatibility). Second, it will require human expertise and manual intervention to generate and maintain such cross-compilation environments, as well as to ensure that the target-specific builds of systems such as *HALE-IoT* actually work without errors (e.g., *Illegal instruction*).

F. Runtime Services Hardcoded to 0.0.0.0: <port>

Sometimes network services (e.g., Web-servers) have the port and the network interface binding hardcoded instead of being read from a configuration file (whether standard or proprietary). This is problematic, as it exposes the original built-in network server to potential attacks coming from attacker-accessible interfaces. In fact, for the 32 firmware in our data set that expose Web services, the interface and/or port was hardcoded directly into the binary. With *HALE-IoT* (cf. Fig. 1) the aim is ideally to isolate (inherently) vulnerable services to

[9]https://github.com/buildroot/buildroot/blob/master/CHANGES

127.0.0.1:<port>, and to expose only the hardened services via *HALE-IoT*.

One possible workaround to this challenge is to manually reverse-engineer and binary-patch the executable files of interest and, ideally, force them to bind to 127.0.0.1:<port>. While this approach will most likely work in most cases, it still cannot scale similarly to the automated configuration change approach we presented above. Another possible workaround is to force-start a dummy TCP/UDP server on 0.0.0.0:<port> before the built-in network service (e.g., Web-server on port 80) has a chance to bind to it, and then observe how the original service behaves for rebinding (e.g., moves to another port, moves to another interface, fails to start altogether). Implementing and testing these adjustments is the scope of our future work.

G. From Self-Signed HTTPS Certificates to Full CA Chains

One of the core aims of *HALE-IoT* is to generically harden IoT devices with proper HTTPS, including support for full-chain certificates. For this, the *HALE-IoT* approach uses the concept of HTTPS-proxying that creates a proper HTTPS service point that is relayed to the built-in webserver. Our current implementation choice is to use SSL-proxy, which provides the latest and most secure TLS implementations, and supports full chain certificates. However, in order to simplify our experiments, and for several practical reasons, we used self-signed certificates generated by the SSL-proxy itself. Should we deploy and evaluate *HALE-IoT* on real-world Internet-facing IoT devices in the future, the following minimal steps would ensure an example implementation when using proper PKI full certificate chains.

1) Configure and connect the device to a public DNS subdomain name under the user's control (e.g., using DDNS services or otherwise), for example https://device-X.fleet-Y.service-provider.c0m.[10]
2) Have a full certificate chain issued by a trusted CA (e.g., Let's Encrypt) and covering https://*.fleet-Y.service-provider.c0m or https://device-X.fleet-Y.service-provider.c0m (depending on the desired granularity of "device identity management" versus "PKI/certificate/key control").
3) Use the corresponding full certificate chain and its private key(s) to configure the SSL-proxy implant that goes into a corresponding device. This can be done before implanting *HALE-IoT*, or while the hardened device is already running by using *HALE-IoT*'s SSH-based administrative interface.

The above is an example of the improvements needed to ensure secure management of DNS, PKI, certificates, private-keys, and device identities.

Moreover, effective and efficient PKI implementations represent an ongoing area of research on its own [65], [66], especially when considering deployment of PKI for IoT [67], [68]. Therefore, we leave research, experimentation, and validation of full-blown PKI support for *HALE-IoT* as future work.

[10]A full first-level DNS domain name would also work for a single device, but is suboptimal and hardly practical for managing larger fleets of devices.

H. Resource Constraints: Static Linking Versus Dynamic Loading

We present two approaches to deal with constraint in physical devices. On the one hand, static linking allows the toolsets within *HALE-IoT* to be self-contained, and not depending on the existence of particular libraries within the target retrofitted firmwares. This makes the approach highly scalable: cross-compile once, run everywhere. However, this approach considerably increases the size of the binaries included with the implant. This is problematic from a storage perspective and from a memory perspective, as there is essentially a possible duplication of library code loaded into RAM due to static linking. Flash storage and RAM memory are quite constrained and minimal in many devices. For instance, Asus RT-N12+ features a 32-MB RAM chipset, where 28 MB is allocated for userland applications, from which *less than* 2.5 MB *was available for the entire HALE-IoT*.

On the other hand, dynamic loading allows the toolsets to be built with minimal binary size and runtime RAM memory consumption, as there is no code duplication and the hardening toolsets can rely on the libraries already present on the device's storage and RAM. However, this approach is highly nonscalable. For example, it means that the toolsets would have to be linked with dynamic loading to a myriad of library versions present in each different firmware version. Even if that could be automated somehow, it does not guarantee that the library exposes the correct and expected interface and functions (e.g., if the library is vendor-customized or missing headers).

This challenge is not easy to solve systematically. Our experience suggests that the best approach is to write highly optimized toolsets designed to fit highly constrained devices. If this is not feasible, then the newly developed hardening toolsets, especially those that are tailored specifically for IoT devices, should incorporate these design principles.

I. Inherent Limitations of WAFs and xAFs

Like any piece of software, WAFs and xAFs or, in general, any implant that can be retrofitted into a system may be subject to limitations that can range from its own implementation errors (e.g., WordPress WAF plugin recently vulnerable to SQLi itself,)[11] or new vulnerabilities and zero-days to outdated software configurations that may arise in the future. Such fundamental limitations would also be inherited by *HALE-IoT*.

One limitation is that most WAFs can detect and prevent input-driven exploits, but very often they are unable to detect and prevent other attacks such as "stored XSS." Another limitation is that WAFs are mostly rule-based, so the presence or absence of specific rules may determine the success or failure of detection/prevention. Also, keeping such rules up to date is another factor that may affect or limit the effectiveness of any given WAF. We have designed *HALE-IoT* with an administrative interface in mind. This feature, if used often and correctly, may help overcome the limitations of outdated rulesets and components. Though technically possible, we leave

[11]https://portswigger.net/daily-swig/wordpress-security-plugin-hide-my-wp-addresses-sql-injection-deactivation-flaws

full automation of updating *HALE-IoT* rule-sets (and other components) as future work.

As a result of our work, we have fixed several bugs in Raptor that have improved the overall reliability of the WAF. The discovery of the bugs and the development of their patches are a relatively modest contribution in itself. However, their discovery underpins the importance and the need for experiments such as ours to expose well-known and widely used software to even more scenarios.

Bugs in Raptor WAF—HTTPS: During the course of the experiments, we detected certain bugs in the way the WAF should work [69]. First, the communication lasted longer than expected even when all the responses had been received, causing, for example, the browser to appear to still be loading the Web page. Inspection of the source code revealed that the socket descriptor was not closing, which caused the connection to remain established. Second, we encountered strange behavior when the WAF was running alongside the SSL-proxy. In this case, when we made POST-type HTTP requests that included some payload, Raptor did not detect them correctly. When instead only the WAF was running instead, it worked as expected with the same request. After closer inspection, we found that the WAF checks whether a request is a Web HTTP request, and if so then the Raptor analyzes it. Generally, most Web clients (e.g., curl or Web browsers) send the headers and the data in the same packet (large chunks of data will be divided into multiple different TCP packets). However, the HTTP libraries of the Go programming language split the request: first send the headers, and then the data itself. Hence, Raptor WAF fails to analyze the data from the subsequent GET/POST request(s). We patched Raptor's code to check the data size of the headers, and then to reassemble the packets before analyzing and proxying the traffic to the destination (i.e., firmware native) Web server.

Bugs in Raptor WAF—Memory Leaks: We found several memory-related problems in Raptor WAF [70], [71]. First, we encountered a `use-after-return` error, that are many times exploitable [72]. This error occurs when a function returns the memory address of a local variable, which is "destroyed" when the function terminates. Therefore, the returned pointer references to an area of the stack that could be used for another function, and could cause unwanted behavior or exploitation of the program [73]. Finally, we found several cases where dynamically allocated memory areas are not properly released, which caused memory leaks. Not freeing up memory causes the program to eventually store more memory than it needs, which is a major issue with memory-constrained devices (see also Section VII-H). This can be an overall limiting factor to the usability of the retrofitting implants, and can also lead to general instability and crashes systems on which Raptor WAF is installed, meaning it can very well affect high-end servers and not just constrained IoT devices. Finally, there is inherent risk in the uninitialized memory created by the dynamic allocation algorithm. This is not a security bug per-se (rather a feature of many programming languages), but certain functions such as `malloc` return a pointer to a block of memory that has uninitialized values and can be potentially exploited [74], [75]. However, the shortcoming of having uninitialized memory areas can be effectively remedied by making use of the `calloc` function, which fills the dynamically allocated memory block with zeros as a deterministic initial value, at the expense of minor performance overheads.

VIII. Related Work

Hardening legacy IoT devices has been a subject of several research papers over the years [14], [20], [22], [77], [80], [105], [106]. At the same time, hardening systems and applications (which also could be extended to IoT at least) have seen a massive body of work on two separate directions, namely, WAF [37], [90], [91], [92], [93], [94], [95] and IDS [83], [84], [85], [86], [87], [88], [89]. Related works, which we showcase here, follow different strategies and we group them into the following categories.

1) Embedding defensive software or retrofitting security measures.
2) Securing firmware from malicious modifications.
3) Securing access control and communications.
4) Web and application firewalls.
5) IDSs.

Table V summarizes the massive body of related works.

A. Retrofitting, Patching, and Hardening for Security

Enhancing the security of a single IoT device is a defensive strategy that works best when the devices are not part of a large centralized network of IoT devices. Cui and Stolfo [20] introduced the notion of SEM, a software design to embed defensive software into an existing installation. The authors embedded an IDS and showed how these strategies can lead to the detection of stealthy malware (i.e., a rootkit) into a Cisco router. Choi et al. [14] followed a similar approach in their research. They developed a scheme to deploy security features in poorly secured IoT devices through the deployment of a monitoring Web service that manages multiple IoT devices in a network. Recently, Maroof et al. [76] presented iRECOVer, a holistic solution for the security management of IoT devices. It aims to replace "vulnerable modules" with "secure modules" and offers "secure channels" for communicating devices, without specifically addressing backward compatibility and intended equivalent functionality of secured modules. The authors demonstrated iRECOVer on a single Raspberry-Pi 4 Model B device running customized open-source Linux distribution. The authors were unable to demonstrate iRECOVer on real-world IoT devices, as they acknowledge that *programming a commodity IoT device is difficult. HALE-IoT* is fundamentally different, as it is demonstrated to work on a large number of heterogeneous and original-commodity IoT devices/firmware, and does not replace original modules, but rather wraps them in added-security layers. These works showed that retrofitting security measures is a process agnostic to the platform (hardware and software), and it does not need to be attuned to any executable format. Similarly to approaches based on SEM, *HALE-IoT* is installed alongside the original operating system and injects protecting payloads into the target. However, prior works propose hardening solutions that are tailored to specific attacks and are limited by scale and lack

TABLE V
SUMMARY RELATED WORK

Category	Reference	Highlight
Retrofitting, Patching, and Hardening for Security	[20]	Introduces the concept of SEM to inject IDS into the embedded device.
	[14]	Proposes a scheme to minimize vulnerabilities and threats, improving the security of IoT devices.
	[76]	Proposes the use of a service-based architecture in order to be able to monitor and replace vulnerable modules on the fly.
	[77]	Introduces DECAF, a system to remove redundant code from UEFI firmware and thus decrease the possibility of vulnerable code.
	[55]	Proposes ABSR, a technique to disable unused firmware features and remove unused binary files.
	[78]	Introduces a system to identify and remove unused basic blocks from the binary code of shared libraries.
	[79]	Designs WebDroid, a framework for building secure embedded web interfaces.
(Malicious) Firmware Modifications	[80]	Analyzes the security impact of untrusted hard drives. The authors analyze the firmware of a hard drive and infect it with a backdoor.
	[55]	Demonstrates that firmware updating is a feature that can be exploited to inject firmware modifications.
	[81]	Analyzes the exposure of PLCs to firmware modifications and the feasibility of this attack through a proof of concept.
	[82]	Analyzes the impact of firmware modification in smart grid environments
Intrusion Detection Systems	[83]	Conducts a survey of IDS focused on IoT environments, as well as the different ML and DL techniques used for attack detection.
	[84]	Proposes an intelligent architecture for IoT based on Complex Event Processing (CEP) and ML to detect IoT attacks in real time.
	[85]	Introduces Passban, an ML-based IDS architecture to detect anomalies in IoT network traffic.
	[86]	Presents a framework that combines ML with software defined networking and network function virtualization technology for the detection of IoT attacks.
	[87]	Proposes an IDS for anomaly detection based on a Multimodal Deep Auto Encoders and a classification of detected attacks based on ML algorithms.
	[88]	Presents Kitsune, an IDS based on autoencoders to detect network attacks that allows learning in an unsupervised way.
	[89]	Proposes an IDS based on deep learning techniques to detect DoS attacks, port scanning or brute force in IoT environments.
Application Firewalls and WAFs	[37]	Conducts a survey with the different existing approaches to secure web applications.
	[90]	Compares the different existing WAF solutions in the literature.
	[91]	Measures the effectiveness of WAFs to prevent injection attacks in web applications.
	[92]	Analyzes the performance of the most used open source WAFs.
	[93]	Proposes WAFFle, a tool to fingerprint the rules of a WAF and craft attacks using some loophole in the filtering rules.
	[94]	Presents an approach based on machine learning algorithms to detect loopholes in the WAF and generate attacks that bypass it.
	[95]	Introduces a prototype that combines static and dynamic verification so that WAFs can formally guarantee the absence of certain types of bugs.
Authentication and Encryption of IoT Communications	[7]	Conducts a large-scale analysis of TLS and SSH servers presenting evidence of the vulnerability of many of their keys.
	[28]	Analyzes TLS deployments in IoT environments, comparing certificates and connection parameters.
	[29]	Presents a study to evaluate TLS vulnerabilities in devices, comparing their results with those found in [28]
	[96]	Conducts a study of the size of the RSA keys used in the most popular protocols, finding that many of them use 512-bit RSA keys.
	[97]	Analyzes TLS network traffic of IoT devices uncovering weaknesses in the way TLS is deployed (e.g., insecure versions, lack of certificate validation).
	[27]	Performs a large-scale measurement of existing certificates in the IPv4 address space, noting that nearly 88% of certificates are invalid.
	[98]	Analyzes the response time, overload, and network latency of the TLS and DTLS protocols in IoT middleware systems used in the e-health ecosystem.
	[99]	Conducts a usability study of the TLS deployment process for HTTPS.
	[100]	Performs a study on the two web security features HSTS and public-key pinning.
Over-The-Air (OTA) Firmware and Software Updates	[101]	Conducts a survey of existing works in the literature related to secure firmware upgrade.
	[102]	Summarizes the key principles of OTA updates for IoT devices.
	[103]	Proposes a security architecture to facilitate the updating of software and hardware in vehicles.
	[104]	Proposes the use of assurance case templates that comply with ISO 26262 and SAE J3061 to guarantee the security of OTA updates.
	[10]	Conducts a survey on OTA updates in vehicles connected to the network.

of automation. *HALE-IoT* is designed as a generic method to deploy universal and hardening solutions with proven effectiveness, while at the same time minimizing intrusion and reconfigurations to the original firmware.

Not every hardening tool is universal, as some are designed to secure a more specific section of IoT devices. For example, Christensen et al. [77] introduced DECAF, a unified extensible firmware interface (UEFI) firmware code pruning system to reduce redundant and possibly vulnerable code in firmware while increasing system performance and security. This firmware commonly exists in motherboards. The DECAF platform does this "debloating" by performing "dynamic iterative surgery" and utilizing existing knowledge of the firmware to remove known possible issues. The authors conclude that in some cases, DECAF reduces the UEFI firmware code by over 70%, thus, notably reducing the attack surface of the firmware. The authors claim that DECAF could potentially be extended to prune any type of firmware. Similarly, Cui et al. [55]

proposed ABSR as an early conceptual ideal firmware code-debloating technique achieved via binary-patching and binary-rewriting. Recently, Zhang et al. [78] presented *μTrimmer*, a system to identify and remove unused basic blocks from binary code of shared libraries and tools. The authors implemented *μTrimmer* for the MIPS architecture (a very common one for IoT devices), and tested its effectiveness on SPEC CPU2017 benchmarks, popular firmware applications (e.g., OpenSSL), and a single real-world wireless router firmware image. *μTrimmer* demonstrated that the challenge of static library debloating on stripped binaries, while being enormous, is not insurmountable for MIPS-based firmware; their system produced functional programs while reducing unnecessary exposed code surface and eliminating various reusable code gadgets. However, debloating itself is ineffective at hardening core services in the firmware (e.g., fragments of the firmware that cannot, or should not, be pruned). Additionally, debloating in principle is a high-risk technique, as it may prune code segments that are instrumental for the normal intended operation of the system/device as a whole. Our system avoids debloating altogether and hardens the potentially vulnerable services with securing wrappers that bring proven effectiveness (e.g., Raptor WAF) and security guarantees (e.g., SSL-proxy).

Standalone IoT devices often interface with the user via built-in Web servers due to its wide and cost-effective adoption. However, Web services often introduce vulnerabilities to the system. Gourdin et al. [79] tackled this issue by developing WebDroid, an IoT focused Android OS Web interface development framework with security as a priority. WebDroid enables developers to easily create more-secure Web interfaces for their Android-based IoT devices. The framework takes into account many important security issues, such as bad authentication practices, XSS, and CSRF. These frameworks are an interesting first step toward securing devices for vendors that lack the means to produce secure environments [107]. However, these types of frameworks are meant to be integrated into the source-code and development life-cycle, and can not be easily adopted to secure firmware already deployed. Our work, on the other hand, practically demonstrates a systematic approach to integrating defensive measures post-deployment and without access to the source-code.

B. Malicious Firmware Modifications

Other related works perform firmware modification to attack devices [80], such as malware targeting USB devices [108] or printers [55], as well as attacks to critical infrastructures [81] (including smart grids [82]). These works are certainly a strong testament that firmware modifications have important real-world implications. However, modifying the firmware to embed defensive and protective mechanisms (as we do with *HALE-IoT*) requires an entire methodological consideration and evaluation to both preserve the correct functioning of the device (Section V-E) and assess the real effectiveness of the multidimensional defenses (Section V-D).

C. Intrusion Detection Systems

IDS have been a subject of research over the last few years. Thakkar and Lohiya [83] conducted a survey on IDS in IoT environments distinguishing between strategies for placing an IDS and for analysis. The most recent related work focuses on building IDSs based on machine learning (ML) [84], [85], [86] and deep learning (DL) [87], [88], [89] models. However, most of them do not take the limitations of legacy IoT devices into account in terms of storage and computing capacity to run the trained models, using at best the Raspberry Pi to evaluate the model in IoT. *HALE-IoT* is a methodology to retrofit security measures in IoT devices, and it is not limited to any tool, allowing the implementation of ML or DL algorithms instead of the WAF used. However, for the implementation of such measures, it is necessary that they adhere to certain assumptions regarding the footprint of the application, dependencies, and architecture of the device.

D. Application Firewalls and WAFs

The security of server-side applications and the main vulnerabilities that affect this type of application has been a subject of research in recent decades [37]. With the growth of the Internet and the services offered through the network, Web applications have become one of the main services attacked. One of the main protections for this type of attack is WAFs, which have been the subject of research in the last decade [90], [91], [92] and are widely used in the industry. WAFs are responsible for monitoring HTTP traffic between users and Web applications, being able to effectively identify known attacks. However, because they are not able to identify zero-day attacks, their signatures need to be updated periodically [37]. Although, there are ways to overcome WAFs (e.g., loopholes in the WAF rules [93], [94]), WAFs can also, in some cases, formally show/guarantee absence of certain bugs. Thus, we assume adding WAF as part of *HALE-IoT* methodology is sound, sane, and increases cybersecurity [95] by protecting device Web interfaces from known attacks that are being exploited automatically on a large scale.

E. Authentication and Encryption of IoT Communications

Perhaps weak authentication, lack of encryption, and vulnerable Web services are altogether one of the largest attack surfaces in IoT devices to date [7], [27], [28], [29], [96], [97], [98]. There is a significant number of previous works that:

1) measures and points out crypto-security deficiencies in the IoT realm as a whole [7];
2) identify the use of weak cryptography in constrained devices [28], [29]; or
3) discover weaknesses in the way TLS or PKI is deployed over the IoT [27], [97].

However, none of the existing works in the literature manage to effectively harden these services due to its intrinsic complexity: “the HTTPS deployment process is far too complex even for people with proficient knowledge in the field” [99], not to mention when such deployment is rooted into an obscure component such as the firmware of a legacy IoT device. It is also

well known that the Web is "large and complicated enough to make even conceptually simple security upgrades challenging to deploy in practice" [100]. The Web of IoTs of all networks is perhaps one of the *hardest to harden*. Still, our work presents a practical, sound, and actionable contribution to addressing these nontrivial challenges.

F. Over-the-Air Firmware and Software Updates

Kolehmainen [101] performed a survey of secure firmware updates for IoT. The author concluded that there are almost as many firmware and software update procedures as there are manufacturers, and proposed a common four-element update model: 1) packing; 2) delivery; 3) authentication; and 4) attestation. Bauwens et al. [102] summarized and outlined key OTA principles for IoT devices and deployments.

Regarding (secure) OTA and Firmware OTA (FOTA) implementations, the automotive industry is perhaps the forerunner and trend-setter in the research literature. Idrees et al. [103] showcased a model for manufacturers, workshops, and vehicles to establish a secure end-to-end link using a trusted platform model and secure communication. The model can be used to secure FOTA updates. Chowdhury et al. [104] proposed an ISO 26262 and SAE J3061 utilizing an assurance case template for OTA updates. If used properly, the template is a valuable tool for manufacturers to root out security issues in their automotive OTA implementation in the development phase.

Halder et al. [10] conducted a survey in regard to OTA updates of network-connected vehicles. They identified some challenges that the industry has yet to fully solve. For example, the software distribution needs to protect privacy as well as be secure. Latency of the software installation can also be an issue, especially for autonomous vehicles. Furthermore, since key management is generally based on the trust of preinstalled keys, key refresh may be in order considering the lifetime of a car.

However, our present work has a different and complementary focus, in that *HALE-IoT* does not propose to solve any challenges faced by (secure) OTA/FOTA software updates. In fact, *HALE-IoT* itself could be delivered/deployed by any OTA/FOTA system that is running (or supported) by the particular device(s). We leave the exploration of integrating *HALE-IoT* into OTA/FOTA workflows as promising future work.

IX. Conclusion

The Internet and private networks are littered with millions of vulnerable IoT devices. A large number of these devices are effectively *abandoned* by manufacturers, who do not issue patches to fix known issues. This prevents users and network administrators from keeping their devices up to date and, thus, poses an endemic risk to the security of the Internet, as well as of the enterprise and private/home networks. Hardening IoT devices allows the attack surface to be reduced, which emerges as a promising countermeasure. However, prior work has limited scope, and clearly fails to deal with the heterogeneity and the many technological constraints of both modern and legacy IoT devices.

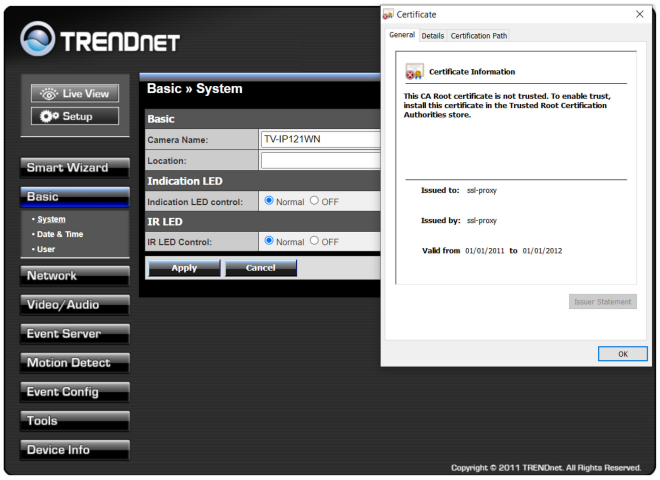

Fig. 6. Screenshot depicting emulated TRENDnet TV-IP121WN IPcam's Web-interface along with the generated HTTPS certificate as part of the SSL/TLS hardening by *HALE-IoT* (emulation).

In this article, we presented a systematic methodology designed to retrofit sophisticated state-of-the-art defensive mechanisms into IoT firmware, with particular focus on legacy and obsolete firmware. We tested our framework with a wide-range of firmware images from different vendors and heterogeneous architectures, totaling 395 emulated firmware and four physical devices. Our results demonstrated that *HALE-IoT* successfully retrofits defensive implants in a scalable and safe manner (i.e., without breaking the firmware). We also evaluated the performance of our approach under a battery of one hundred attacks, showing it is feasible to deploy *HALE-IoT* in the wild.

We discussed our findings and identified a number of limitations that showed the challenges and the idiosyncrasies of hardening IoT devices. Our discussion also elicited a number of future promising directions. First, an interesting avenue of research is to explore the use of defense-in-depth strategies as a mechanism to harden IoT devices. This introduces non-negligible tradeoffs between the complexity of the method (e.g., iptables, Snort, and fail2ban) and the overall gain. Second, we identify the need to automate the cross-compilation of the implants to more CPU architectures (e.g., RISC-V and Xtensa) and to a more diverse set of obsolete environments (e.g., MIPS-I), while minimizing the overall footprints at build and runtime (e.g., storage, RAM, and CPU). Finally, we would like to encourage researcher and industry practitioners having access to large sets of physical devices to enlarge evaluation and support in *HALE-IoT*.

Appendix

A. Supporting Materials—Screenshots

1) Screenshots for Evaluation on IPTV Camera (Emulation): Fig. 6 shows the TRENDnet TV-IP121WN IP camera Web interface running with *HALE-IoT* via firmware emulation.

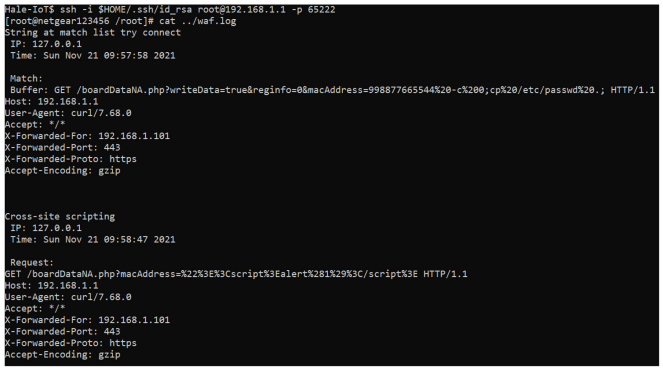

Fig. 7. Evaluation on CVE-2016-1555 (emulation): Screenshot depicting successful detection and prevention of both XSS and CI attacks attempting to exploit CVE-2016-1555 on an emulated firmware hardened with *HALE-IoT*.

Fig. 8. Evaluation on RPi3 with OpenWrt (device): Screenshot depicting XSS in OpenWrt's LuCI Web interface running on RPi3 without *HALE-IoT*.

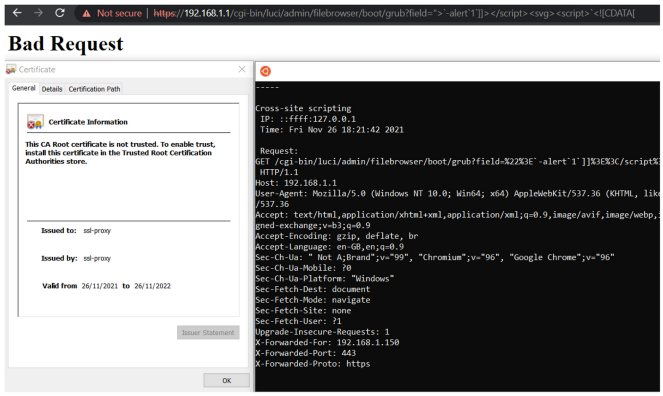

Fig. 9. Evaluation on RPi3 with OpenWrt (device): Screenshot depicting successful prevention of XSS and addition of HTTPS after RPi3 running the same originally vulnerable OpenWrt was hardened with *HALE-IoT*.

2) Screenshots for Evaluation on CVE-2016-1555 (Emulation): Fig. 7 shows the successful detection of the attempted exploitation of CVE-2016-1555 (XSS and CI) on an emulated firmware that is hardened by *HALE-IoT*.

3) Screenshots for Evaluation on RPi3 With OpenWrt (Device): Figs. 8 and 9 show how *HALE-IoT* works on a RaspberryPi 3 running OpenWrt (Section VI-B). Fig. 8 shows the successful exploitation of an XSS in the LuCI Web interface without *HALE-IoT*, while Fig. 9 shows that the attack is detected and prevented when the device is hardened by *HALE-IoT*.

4) Screenshots for Evaluation on Several Devices (Device): Figs. 10–12 show the successful installation of *HALE-IoT* on the physical devices Asus RT-N12+ B1 (Section VI-C), Netgear R6220 (Section VI-D), and Linksys EA4500 (Section VI-E), respectively.

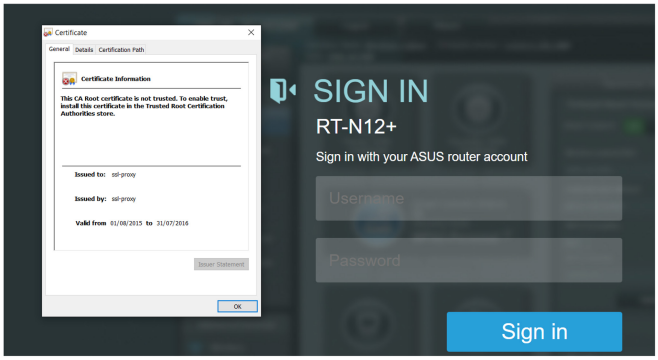

Fig. 10. Evaluation on Asus RT-N12+ (device): Screenshot depicting *HALE-IoT* implant successfully running on Asus RT-N12+ B1 device.

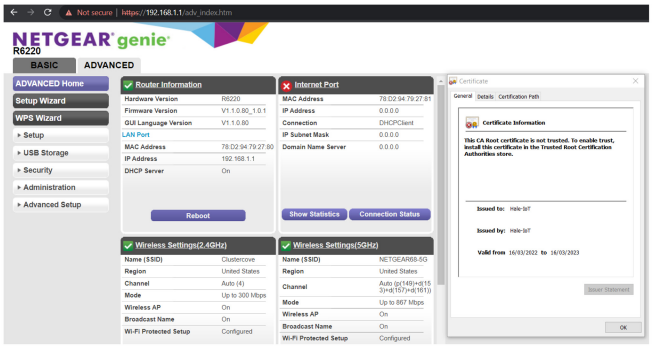

Fig. 11. Screenshot depicting Netgear R6220's Web-interface along with the generated HTTPS certificate as part of the SSL/TLS hardening by *HALE-IoT*.

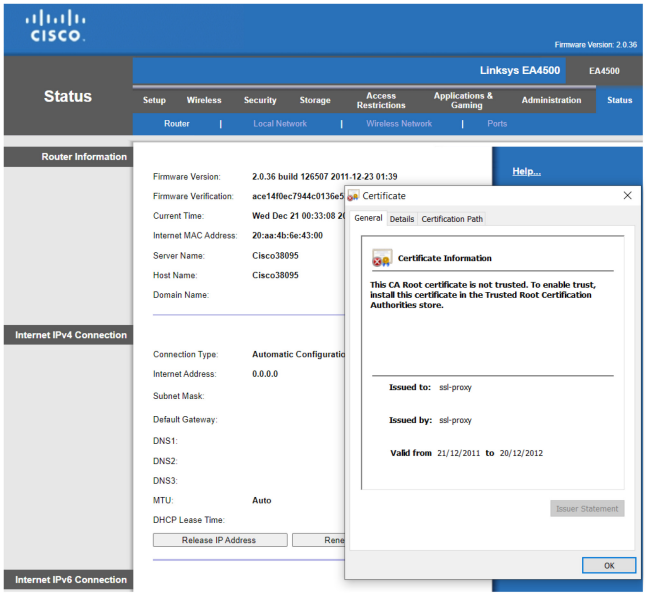

Fig. 12. Screenshot depicting Linksys EA4500's Web-interface along with the generated HTTPS certificate as part of the SSL/TLS hardening by *HALE-IoT*.

5) Screenshots for Evaluation on Linksys wrtsl54gs Hardcoded Binary (Emulation): Fig. 13 shows the Ghidra decompiled code of the BusyBox httpd applet showing that it accepts an option to change the IP address to bind the Web server. On the other hand, Fig. 14 shows that this option is not available in the help menu of the httpd applet. Finally, Fig. 15 shows that by using this hidden "$-l$" option it is possible to

Fig. 13. Linksys wrtsl54gs Hardcoded Binary (emulation): Screenshot depicting the presence of a hidden "$-l$" option used for binding network interface.

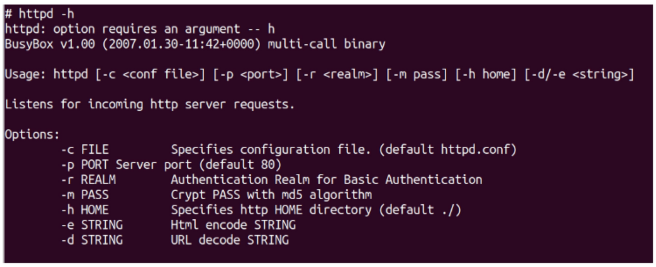

Fig. 14. Linksys wrtsl54gs Hardcoded Binary (emulation): Screenshot depicting that the builtin *httpd* server's help menu does not normally show the hidden "$-l$" option.

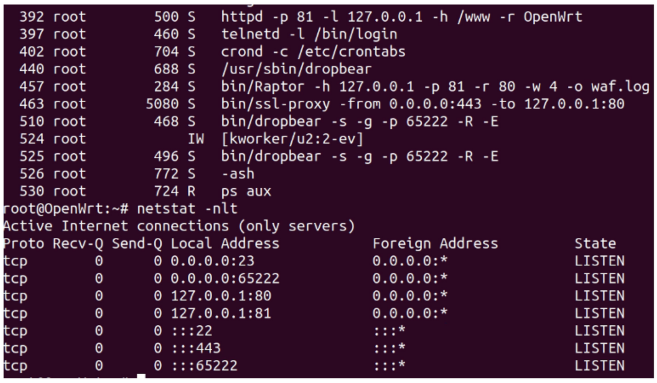

Fig. 15. Linksys wrtsl54gs Hardcoded Binary (emulation): Screenshot depicting that builtin *httpd* server did a successful rebind to 127.0.0.1 by forcing this via the hidden "$-l$" option.

isolate from 0.0.0.0 the (potentially vulnerable) Web server when it is working with *HALE-IoT* (Section VI-A).

List of Acronyms

ABI	Application Binary Interface.
ABSR	Autotomic Binary Structure Randomization.
AFL	American Fuzzy Lop.
BANG	BinaryAnalysis-NG.
CA	Certificate Authority.
CI	Command Injection.
CIS	Center for Internet Security.
CPU	Central Processing Unit.
CSRF	Cross-Site Request Forgery.
CVE	Common Vulnerabilities and Exposures.
DDNS	Dynamic Domain Name System.
DFA	Deterministic Finite Automata.
DL	Deep Learning.
DNS	Domain Name System.
EOL	End-Of-Life.
FMK	Firmware-Mod-Kit.
FOTA	Firmware Over-The-Air.
FTP	File Transfer Protocol.
HSTS	HTTP Strict-Transport-Security.
HTTP	HyperText Transfer Protocol.
HTTPS	HyperText Transfer Protocol Secure.
IDS	Intrusion Detection System.
IoT	Internet-of-Things.
IP	Internet Protocol.
ISA	Instruction Set Architecture.
ISO	International Organization for Standardization.
JFFS	Journaling Flash File System.
LAN	Local Area Network.
MCU	MicroController Unit.
MITM	Man In The Middle.
ML	Machine Learning.
MQTT	Message Queue Telemetry Transport.
OS	Operative System.
OTA	Over-The-Air.
PC	Personal Computer.
PCRE	Perl-Compatible Regular Expressions.
PKI	Public key infrastructure.
QEMU	Quick EMUlator.
RAM	Random Access Memory.
RCE	Remote Code Execution.
SAE	Society of Automotive Engineers.
SEM	symbiotic embedded machine.
SFTP	Secure File Transfer Protocol.
SPI	Serial Peripheral Interface.
SQLi	Structured Query Language injection.
SquashFS	Squash File System.
SSH	Secure Shell.
SSL	Secure Socket Layer.
TCP	Transmission Control Protocol.
Telnet	Teletype Network.
TLS	Transport Layer Security.
UDP	User Datagram Protocol.
UEFI	Unified Extensible Firmware Interface.
UPnP	Universal Plug and Play.
USB	Universal Serial Bus.
VPN	Virtual Private Network.
WAF	Web Application Firewall.
WAN	Wide Area Network.
WLAN	wireless Local Area Network.
XSS	Cross-Site Scripting.

ACKNOWLEDGMENT

The authors would like to thank Andrei Costin for hosting Javier Carrillo-Mondéjar at the University of Jyväskylää, and "call for stays at universities and research centers abroad for the year 2021" from University of Castilla-La Mancha for supporting this research visit. Hannu Turtiainen also thanks the Finnish Cultural Foundation/Suomen Kulttuurirahasto (https://skr.fi/en) for supporting his Ph.D. dissertation work and research (under grant decision no. 00221059) and the Faculty of Information Technology of the University of Jyväskylä (JYU), in particular, Prof. Timo Hämäläinen, for partly supporting and supervising his Ph.D. work at JYU in 2021–2023. The authors thank Jami Laamanen for his contributions during the early stages of the experiments [109]. The authors also acknowledge the use of royalty-free icons courtesy of www.flaticon.com (icons by: Cuputo, Good Ware, rukanicon, and Prosymbols). Last but not least, the authors would like to thank the anonymous reviewers and journal editors for their valuable feedback and insightful comments that meaningfully improved this article.

REFERENCES

[1] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A large-scale analysis of the security of embedded firmwares," in *Proc. 23rd USENIX Security Symp.*, 2014, pp. 95–110.

[2] A. Costin, A. Zarras, and A. Francillon, "Automated dynamic firmware analysis at scale: A case study on embedded Web interfaces," in *Proc. 11th ACM Asia Conf. Comput. Commun. Security*, 2016, pp. 437–448.

[3] D. Chen, M. Egele, M. Woo, and D. Brumley, "Towards automated dynamic analysis for linux-based embedded firmware," in *Proc. NDSS*, vol. 25, 2016, pp. 1–8.

[4] M. Antonakakis et al., "Understanding the mirai botnet," in *Proc. 26th {USENIX} Security Symp.*, 2017, pp. 1093–1110. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis

[5] A. Cui and S. J. Stolfo, "A quantitative analysis of the insecurity of embedded network devices: Results of a wide-area scan," in *Proc. 26th Annu. Comput. Security Appl. Conf. (ACSAC)*, 2010, pp. 97–106.

[6] Carna Botnet, "Internet census 2012: Port scanning/0 using insecure embedded devices," SourceForge, San Diego, CA, USA, White Paper, 2012.

[7] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman, "Mining your ps and Qs: Detection of widespread weak keys in network devices," in *Proc. 21st {USENIX} Security Symp.*, 2012, pp. 205–220.

[8] A. Mirian et al., "An Internet-wide view of ICS devices," in *Proc. IEEE 14th Annu. Conf. Privacy Security Trust (PST)*, 2016, pp. 96–103.

[9] D. Kumar et al., "All things considered: An analysis of IoT devices on home networks," in *Proc. 28th {USENIX} Security Symp.*, 2019, pp. 1169–1185.

[10] S. Halder, A. Ghosal, and M. Conti, "Secure over-the-air software updates in connected vehicles: A survey," *Comput. Netw.*, vol. 178, Sep. 2020, Art. no. 107343.

[11] O. Alrawi et al., "The circle of life: A large-scale study of the IoT malware lifecycle," in *Proc. 30th USENIX Security Symp.*, 2021, pp. 3505–3522. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/alrawi-circle

[12] R. Millman. "IoT Devices Are More Vulnerable Than Ever." 2021. [Online]. Available: https://www.itpro.co.uk/network-internet/internet-of-things-iot/360850/iot-devices-are-more-vulnerable-than-ever

[13] S. Weston. "83 Million IoT Devices at Risk of Hacking." 2021. [Online]. Available: https://www.itpro.co.uk/network-internet/internet-of-things-iot/360612/83-million-iot-devices-at-risk-of-hacking

[14] S.-K. Choi, C.-H. Yang, and J. Kwak, "System hardening and security monitoring for IoT devices to mitigate IoT security vulnerabilities and threats," *KSII Trans. Internet Inf. Syst.*, vol. 12, no. 2, pp. 906–918, 2018.

[15] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically hardening Web applications using precise tainting," in *Proc. IFIP Int. Inf. Security Conf.*, 2005, pp. 295–308.

[16] T. Fraser, L. Badger, and M. Feldman, "Hardening COTS software with generic software wrappers," in *Proc. IEEE Symp. Security Privacy*, 1999, pp. 2–16.

[17] S. Kubler, K. Främling, and A. Buda, "A standardized approach to deal with firewall and mobility policies in the IoT," *Pervasive Mobile Comput.*, vol. 20, pp. 100–114, Jul. 2015.

[18] N. Gupta, V. Naik, and S. Sengupta, "A firewall for Internet of Things," in *Proc. IEEE 9th Int. Conf. Commun. Syst. Netw. (COMSNETS)*, 2017, pp. 411–412.

[19] B. P. Sindhuri and M. K. Rao, "IoT security through Web application firewall," *Int. J. Eng. Technol.*, vol. 7, p. 58, Mar. 2018.

[20] A. Cui and S. J. Stolfo, "Defending embedded systems with software symbiotes," in *Proc. Int. Workshop Recent Adv. Intrusion Detection*, 2011, pp. 358–377.

[21] C. Ye, P. P. Indra, and D. Aspinall, "Retrofitting security and privacy measures to smart home devices," in *Proc. IEEE 6th Int. Conf. Internet Things Syst. Manag. Security (IOTSMS)*, 2019, pp. 283–290.

[22] C. Frank, C. Nance, S. Jarocki, and W. E. Pauli, "Protecting IoT from Mirai botnets; IoT device hardening," *J. Inf. Syst. Appl. Res.*, vol. 11, pp. 33–44, Aug. 2018.

[23] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices," in *Proc. NDSS*, 2018, pp. 1–8. [Online]. Available: http://s3.eurecom.fr/docs/ndss18_muench.pdf

[24] D. Stuttard and M. Pinto, *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*. Hoboken, NJ, USA: Wiley, 2011.

[25] H. Bojinov, E. Bursztein, E. Lovett, and D. Boneh, "Embedded management interfaces: Emerging massive insecurity," in *Proc. BlackHat USA*, 2009, pp. 1–8.

[26] M. Dahlmanns, J. Lohmöller, J. Pennekamp, J. Bodenhausen, K. Wehrle, and M. Henze, "Missed opportunities: Measuring the untapped TLS support in the industrial Internet of Things," in *Proc. 17th ACM Asia Conf. Comput. Commun. Security*, 2022, pp. 252–266.

[27] T. Chung et al., "Measuring and applying invalid SSL certificates: The silent majority," in *Proc. Internet Meas. Conf.*, 2016, pp. 527–541.

[28] N. Samarasinghe and M. Mannan, "Short paper: TLS ecosystems in networked devices vs. Web servers," in *Proc. Int. Conf. Financial Cryptography Data Security*, 2017, pp. 533–541.

[29] N. Samarasinghe and M. Mannan, "Another look at TLS ecosystems in networked devices vs. Web servers," *Comput. Security*, vol. 80, pp. 1–13, Jan. 2019.

[30] CoolerVoid. "Raptor_Waf." 2021. [Online]. Available: https://github.com/CoolerVoid/raptor_waf

[31] G. Luptak and D. Palotay. "IoT Botnet Report 2021: Malware and Vulnerabilities Targeted." 2021. [Online]. Available: https://cujo.com/iot-botnet-report-2021-malware-and-vulnerabilities-targeted/

[32] A. Costin and J. Zaddach. "IoT malware: Comprehensive survey, analysis framework and case studies." BlackHat USA. 2018. Accessed: Mar. 2023. [Online]. Available: http://firmware.re/malw/bh18us_costin.pdf

[33] E. Cozzi, M. Graziano, Y. Fratantonio, and D. Balzarotti, "Understanding Linux malware," in *Proc. IEEE Symp. Security Privacy (SP)*, 2018, pp. 161–175.

[34] E. Cozzi, P.-A. Vervier, M. Dell'Amico, Y. Shen, L. Bilge, and D. Balzarotti, "The tangled genealogy of IoT malware," in *Proc. Annu. Comput. Security Appl. Conf.*, 2020, pp. 1–16.

[35] B. Zhao et al., "A large-scale empirical study on the vulnerability of deployed IoT devices," *IEEE Trans. Depend. Secure Comput.*, vol. 19, no. 3, pp. 1826–1840, May/Jun. 2022.

[36] J. Carrillo-Mondéjar, J. Martínez, and G. Suarez-Tangil, "Characterizing Linux-based malware: Findings and recent trends," *Future Gener. Comput. Syst.*, vol. 110, pp. 267–281, Sep. 2020.

[37] X. Li and Y. Xue, "A survey on server-side approaches to securing Web applications," *ACM Comput. Surveys*, vol. 46, no. 4, pp. 1–29, 2014.

[38] A. Cui. "The Overlooked Problem of 'N-day' Vulnerabilities." 2018. [Online]. Available: https://www.darkreading.com/vulnerabilities-threats/the-overlooked-problem-of-n-day-vulnerabilities

[39] S. Vasile, D. Oswald, and T. Chothia, "Breaking all the things—A systematic survey of firmware extraction techniques for IoT devices," in *Proc. Int. Conf. Smart Card Res. Adv. Appl.*, 2018, pp. 171–185.

[40] T. Gilb and S. Finzi, *Principles of Software Engineering Management*, vol. 11. Reading, MA, USA: Addison-Wesley, 1988.

[41] S. Kumar. "Suyashkumar/SSL-Proxy." 2021. [Online]. Available: https://github.com/suyashkumar/ssl-proxy

[42] "Home—Lighttpd—Fly light." Accessed: Mar. 22, 2022. [Online]. Available: https://www.lighttpd.net/

[43] "Let's encrypt." Accessed: Nov. 15, 2021. [Online]. Available: https://letsencrypt.org/

[44] J. Aas et al., "Let's encrypt: An automated certificate authority to encrypt the entire Web," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2019, pp. 2473–2487.

[45] "Binwalk." ReFirmLabs. Accessed: Mar. 27, 2023. [Online]. Available: https://github.com/ReFirmLabs/binwalk.

[46] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice—Automatic detection of authentication bypass vulnerabilities in binary firmware," in *NDSS*, vol. 24, 2015, pp. 1–18.

[47] "Accelerating deployment of arm-based solutions." Accessed: Nov. 18, 2021. [Online]. Available: https://www.linaro.org/

[48] "Linux toolchain—MIPS." Accessed: Nov. 18, 2021. [Online]. Available: https://www.mips.com/develop/tools/compilers/linux-toolchain/

[49] "Buildroot—Making embedded Linux easy." Accessed: Nov. 18, 2021. [Online]. Available: https://buildroot.org/

[50] "MUSL LIBC." Accessed: Nov. 18, 2021. [Online]. Available: https://musl.libc.org/

[51] L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect's Perspective*. London, U.K.: Addison-Wesley Prof., 2015.

[52] C. Heistand et al., "DevOps for spacecraft flight software," in *Proc. IEEE Aerosp. Conf.*, 2019, pp. 1–16.

[53] "Payloadbox/sql-Injection-Payload-List." 2021. [Online]. Available: https://github.com/payloadbox/sql-injection-payload-list

[54] "Payloadbox/Xss-Payload-List." 2021. [Online]. Available: https://github.com/payloadbox/xss-payload-list

[55] A. Cui, M. Costello, and S. Stolfo, "When firmware modifications attack: A case study of embedded exploitation," in *Proc. NDSS*, vol. 22, 2013, pp. 1–8.

[56] A. Hemel. "Using ELF symbols extracted from dynamically linked ELF binaries for fingerprinting." 2021. Accessed: Mar. 2023. [Online]. Available: https://www.tdcommons.org/dpubs_series/4441/

[57] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2017, pp. 363–376.

[58] J. Collake and C. Heffner. "Firmware modification kit." 2013. Accessed: Mar. 2023. [Online]. Available: https://github.com/rampageX/firmware-mod-kit/

[59] D. Giese. "How-to Modify ARM Cortex-M Based Firmware: A Step-by-Step Approach for Xiaomi IoT Devices." 2018. [Online]. Available: https://dontvacuum.me/talks/DEFCON26-IoT-Village/DEFCON26-IoT-Village_How_to_Modify_Cortex_M_Firmware-Xiaomi.pdf

[60] D. Goodin. "A Vigilante Is Putting a Huge Amount of Work Into Infecting IoT Devices." 2017. [Online]. Available: https://arstechnica.com/information-technology/2017/04/a-vigilante-is-putting-huge-amount-of-work-into-infecting-iot-devices/

[61] O. Shwartz, Y. Mathov, M. Bohadana, Y. Elovici, and Y. Oren, "Opening pandora's box: Effective techniques for reverse engineering IoT devices," in *Proc. Int. Conf. Smart Card Res. Adv. Appl.*, 2017, pp. 1–121.

[62] J. Zaddach et al., "AVATAR: A framework to support dynamic security analysis of embedded systems' Firmwares," in *Proc. NDSS*, vol. 23, 2014, pp. 1–9.

[63] A. Hemel. "Binaryanalysis-NG." Accessed: Mar. 27, 2023. [Online]. Available: https://github.com/armijnhemel/binaryanalysis-ng

[64] I. Skochinsky, "Intro to embedded reverse engineering for pc reversers," in *Proc. REcon Conf.*, 2010, pp. 1–8.

[65] P. Gutmann, "PKI: It's not dead, just resting," *Computer*, vol. 35, no. 8, pp. 41–49, 2002.

[66] C. Adams and S. Lloyd, *Understanding PKI: Concepts, Standards, and Deployment Considerations*. Boston, MA, USA: Addison-Wesley Prof., 2003.

[67] Z.-K. Zhang, M. C. Y. Cho, and S. Shieh, "Emerging security threats and countermeasures in IoT," in *Proc. 10th ACM Symp. Inf. Comput. Commun. Security*, 2015, pp. 1–6.

[68] D. Diaz-Sanchez, A. Marín-Lopez, F. A. Mendoza, P. A. Cabarcos, and R. S. Sherratt, "TLS/PKI challenges and certificate pinning techniques for IoT and M2M secure communications," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 4, pp. 3502–3531, 4th Quart., 2019.

[69] "Raptor_Waf issues—Raptor WAF with SSL proxy." Accessed: Nov. 21, 2022. [Online]. Available: https://github.com/CoolerVoid/raptor_waf/issues/15

[70] "Raptor_Waf issues—Stack use-after-return." Accessed: Nov. 21, 2022. [Online]. Available: https://github.com/CoolerVoid/raptor_waf/issues/13

[71] "Raptor_Waf issues—Unfreed memory blocks." Accessed: Nov. 21, 2022. [Online]. Available: https://github.com/CoolerVoid/raptor_waf/issues/14

[72] A. Samsonov and K. Serebryany. "New features in address-sanitizer." 2013. Accessed: Mar. 2023. [Online]. Available: https://llvm.org/devmtg/2013-11/slides/Serebryany-ASAN.pdf

[73] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with WIT," in *Proc. IEEE Symp. Security Privacy*, 2008, pp. 263–277.

[74] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Efficient techniques for comprehensive protection from memory error exploits," in *Proc. USENIX Security Symp.*, vol. 10, 2005, pp. 1251398–1251415.

[75] A. Milburn, H. Bos, and C. Giuffrida, "Safelnit: Comprehensive and practical mitigation of uninitialized read vulnerabilities," in *Proc. NDSS*, vol. 17, 2017, pp. 1–15.

[76] U. Maroof, A. Shaghaghi, R. Michelin, and S. Jha, "iRECOVer: Patch your IoT on-the-fly," *Future Gener. Comput. Syst.*, vol. 132, pp. 178–193, Jul. 2022.

[77] J. Christensen, I. M. Anghel, R. Taglang, M. Chiroiu, and R. Sion, "{DECAF}: Automatic, adaptive de-bloating and hardening of {COTS} firmware," in *Proc. 29th {USENIX} Security Symp.*, 2020, pp. 1713–1730.

[78] H. Zhang, M. Ren, Y. Lei, and J. Ming, "One size does not fit all: Security hardening of MIPS embedded systems via static binary debloating for shared libraries," in *Proc. 27th ACM Int. Conf. Architect. Support Program. Lang. Oper. Syst.*, 2022, pp. 255–270.

[79] B. Gourdin, C. Soman, H. Bojinov, and E. Bursztein, "Toward secure embedded Web interfaces," in *Proc. 20th USENIX Security Symp.*, 2011, pp. 1–8.

[80] J. Zaddach et al., "Implementation and implications of a stealth hard-drive backdoor," in *Proc. 29th Annu. Comput. Security Appl. Conf. (ACSAC)*, 2013, pp. 279–288.

[81] Z. Basnight, J. Butts, J. Lopez, Jr., and T. Dube, "Firmware modification attacks on programmable logic controllers," *Int. J. Crit. Infrastruct. Protect.*, vol. 6, no. 2, pp. 76–84, 2013.

[82] C. Konstantinou and M. Maniatakos, "Impact of firmware modification attacks on power systems field devices," in *Proc. IEEE Int. Conf. Smart Grid Commun. (SmartGridComm)*, 2015, pp. 283–288.

[83] A. Thakkar and R. Lohiya, "A review on machine learning and deep learning perspectives of IDS for IoT: Recent updates, security issues, and challenges," *Arch. Comput. Methods Eng.*, vol. 28, no. 4, pp. 3211–3243, 2021.

[84] J. Roldán, J. Boubeta-Puig, J. L. Martínez, and G. Ortiz, "Integrating complex event processing and machine learning: An intelligent architecture for detecting IoT security attacks," *Exp. Syst. Appl.*, vol. 149, Jul. 2020, Art. no. 113251.

[85] M. Eskandari, Z. H. Janjua, M. Vecchio, and F. Antonelli, "Passban IDS: An intelligent anomaly-based intrusion detection system for IoT edge devices," *IEEE Internet Things J.*, vol. 7, no. 8, pp. 6882–6897, Aug. 2020.

[86] M. Bagaa, T. Taleb, J. B. Bernabe, and A. Skarmeta, "A machine learning security framework for IoT systems," *IEEE Access*, vol. 8, pp. 114066–114077, 2020.

[87] G. Bovenzi, G. Aceto, D. Ciuonzo, V. Persico, and A. Pescapé, "A hierarchical hybrid intrusion detection approach in IoT scenarios," in *Proc. GLOBECOM IEEE Global Commun. Conf.*, 2020, pp. 1–7.

[88] Y. Mirsky, T. Doitshman, Y. Elovici, and A. Shabtai, "KitSune: An ensemble of autoencoders for online network intrusion detection," 2018, *arXiv:1802.09089*.

[89] Y. Otoum, D. Liu, and A. Nayak, "DL-IDS: A deep learning–based intrusion detection framework for securing IoT," *Trans. Emerg. Telecommun. Technol.*, vol. 33, no. 3, 2022, Art. no. e3803.

[90] A. Razzaq, A. Hur, S. Shahbaz, M. Masood, and H. F. Ahmad, "Critical analysis on Web application firewall solutions," in *Proc. IEEE 11th Int. Symp. Auton. Decentralized Syst. (ISADS)*, 2013, pp. 1–6.

[91] H. Holm and M. Ekstedt, "Estimates on the effectiveness of Web application firewalls against targeted attacks," *Inf. Manag. Comput. Security*, vol. 21, no. 4, pp. 250–265, 2013.

[92] S. Prandl, M. Lazarescu, and D.-S. Pham, "A study of Web application firewall solutions," in *Proc. Int. Conf. Inf. Syst. Security*, 2015, pp. 501–510.

[93] I. Schmitt and S. Schinzel, "WAFFle: Fingerprinting filter rules of Web application firewalls," in *Proc. WOOT*, 2012, pp. 34–40.

[94] D. Appelt, C. D. Nguyen, A. Panichella, and L. C. Briand, "A machine-learning-driven evolutionary approach for testing Web application firewalls," *IEEE Trans. Rel.*, vol. 67, no. 3, pp. 733–757, Sep. 2018.
[95] L. Desmet, F. Piessens, W. Joosen, and P. Verbaeten, "Bridging the gap between Web application firewalls and Web applications," in *Proc. 4th ACM Workshop Formal Methods Security*, 2006, pp. 67–77.
[96] L. Valenta, S. Cohney, A. Liao, J. Fried, S. Bodduluri, and N. Heninger, "Factoring as a service," in *Proc. Int. Conf. Financial Cryptography Data Security*, 2016, pp. 321–338.
[97] M. T. Paracha, D. J. Dubois, N. Vallina-Rodriguez, and D. R. Choffnes, "IoTLS: Understanding TLS usage in consumer IoT devices," in *Proc. Internet Meas. Conf.*, 2021, pp. 165–178.
[98] R. T. Tiburski, L. A. Amaral, E. de Matos, D. F. de Azevedo, and F. Hessel, "Evaluating the use of TLS and DTLS protocols in IoT middleware systems applied to E-health," in *Proc. 14th IEEE Annu. Consum. Commun. Netw. Conf. (CCNC)*, 2017, pp. 480–485.
[99] K. Krombholz, W. Mayer, M. Schmiedecker, and E. Weippl, "I have no idea what i'm doing'—On the usability of deploying {HTTPS}," in *Proc. 26th {USENIX} Security Symp.*, 2017, pp. 1339–1356.
[100] M. Kranch and J. Bonneau, "Upgrading HTTPS in mid-air," in *Proc. NDSS*, vol. 24, 2015, pp. 1–9.
[101] A. Kolehmainen, "Secure firmware updates for IoT: A survey," in *Proc. IEEE Int. Conf. Internet Things (iThings) IEEE Green Comput. Commun. (GreenCom) IEEE Cyber Phys. Soc. Comput. (CPSCom) IEEE Smart Data (SmartData)*, 2018, pp. 112–117.
[102] J. Bauwens, P. Ruckebusch, S. Giannoulis, I. Moerman, and E. De Poorter, "Over-the-air software updates in the Internet of Things: An overview of key principles," *IEEE Commun. Mag.*, vol. 58, no. 2, pp. 35–41, Feb. 2020.
[103] M. S. Idrees, H. Schweppe, Y. Roudier, M. Wolf, D. Scheuermann, and O. Henniger, "Secure automotive on-board protocols: A case of over-the-air firmware updates," in *Proc. Int. Workshop Commun. Technol. Veh.*, 2011, pp. 224–238.
[104] T. Chowdhury et al., "Safe and secure automotive over-the-air updates," in *Proc. Int. Conf. Comput. Safety Rel. Security*, 2018, pp. 172–187.
[105] Z. Grimmett, J. Staggs, and S. Shenoi, "Retrofitting mobile devices for capturing memory-resident malware based on system side-effects," in *Proc. IFIP Int. Conf. Digit. Forensics*, 2019, pp. 59–72.
[106] C. Segarra, R. Delgado-Gonzalo, and V. Schiavoni, "MQT-TZ: Hardening IoT brokers using ARM TrustZone:(practical experience report)," in *Proc. IEEE Int. Symp. Rel. Distrib. Syst. (SRDS)*, 2020, pp. 256–265.
[107] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose, "SoK: Security evaluation of home-based IoT deployments," in *Proc. IEEE Symp. Security Privacy*, 2019, pp. 1362–1380.
[108] Cybersecurity. "Security Tip (ST08-001)—Using Caution With USB Drives." 2019. [Online]. Available: https://us-cert.cisa.gov/ncas/tips/ST08-001
[109] J. Laamanen, "Hardening legacy IoT-devices by retrofitting security measures," M.S. thesis, Fac. Inf. Technol., Univ. Jyvaskyla, Jyvaskyla, Finland, 2019.

Javier Carrillo-Mondéjar received the B.Sc. and M.Sc. degrees in computer science and engineering and the Ph.D. degree in advanced computing technologies from the University of Castilla-La Mancha, Ciudad Real, Spain, in 2016, 2017, and 2022, respectively.

He joined the High-Performance Networks and Architectures Group, Informatics Research Institute of Albacete, University of Castilla-La Mancha, as a Research Assistant in 2016. He has also been a Visiting Researcher with King's College London, London, U.K., for five months and the University of Jyväskylä, Jyväskylä, Finland, for three months. His research interests are related to malware detection and classification techniques, with a particular focus on IoT/firmware cybersecurity.

Hannu Turtiainen received the B.Sc. degree in electronics engineering from the University of Applied Sciences, Jyväskylä, Finland, in 2012, the M.Sc. degree in cybersecurity from the University of Jyväskylä, Jyväskylä, in 2020, where he is currently pursuing the Ph.D. degree in software and communication technology.

His research topic is Machine Learning and Artificial Intelligence in the Cybersecurity and Digital Privacy field. He is also working in the IoT field as a Cybersecurity and Software Engineer in Binare.io, Jyväskylä, a deep-tech cybersecurity spin-off from the University of Jyväskylä.

Andrei Costin received the Ph.D. degree from EURECOM/Telecom ParisTech, Sophia Antipolis, France, under co-supervision of Prof. Francilon and Prof. Balzarotti in 2015.

He is currently a Senior Lecturer/Assistant Professor of Cybersecurity with the University of Jyväskylä (Central Finland), Jyväskylä, Finland, with a particular focus on IoT/firmware cybersecurity and Digital Privacy. He has been publishing and presenting at more than 45 top international cybersecurity venues, both academic, such as Usenix Security and ACM ASIACCS, and industrial, such as BalckHat, CCC, and HackInTheBox. He has authored the first practical ADS-B attacks (BlackHat 2012) and has literally established the large-scale automated firmware analysis research areas (Usenix Security 2014)—these two works are considered seminal in their respective areas, being also most cited at the same time. Dr. Costin is also the CEO/co-founder of Binare.io, a deep-tech cybersecurity spin-off from University of Jyväskylä, focused on innovation and tech-transfer related to IoT cybersecurity.

Jose Luis Martínez received the M.Sc. and Ph.D. degrees in computer science and engineering from the University of Castilla-La Mancha, Ciudad Real, Spain, in 2007 and 2009, respectively.

He joined the Department of Computer Engineering at the University of Castilla-La Mancha, in 2005, where he was a Researcher with the Computer Architecture and Technology Group, Albacete Research Institute of Informatics. In 2010, he joined the Department of Computer Architecture, Complutense University in Madrid, Madrid, Spain, where he was an Assistant Lecturer. In 2011, he rejoined the Department of Computer Engineering, University of Castilla-La Mancha, where he is currently a Full Professor. He has also been a Visiting Researcher with Florida Atlantic University, Boca Raton, FL, USA, and the Centre for Communication System Research, University of Surrey, Guildford, U.K. He has over 100 publications in these areas in international refereed journals and conference proceedings. His research interests include video coding and transcoding, and topics related to security.

Guillermo Suarez-Tangil received the M.Sc. and Ph.D. degrees in computer science from Universidad Carlos III de Madrid, Getafe, Spain.

He is currently an Assistant Professor with IMDEA Networks Institute, Leganés, Spain. He is with IMDEA Networks Institute, Madrid, Spain. His research focuses on systems security and malware analysis and detection. In particular, his area of expertise lies in the study of smart malware, ranging from the detection of advanced obfuscated malware to the automated analysis of targeted malware. Before joining IMDEA, he was Lecturer with King's College London, London, U.K., and before that was a Senior Research Associate with the University College London, London, where he was also actively involved in other research areas involved with detecting and preventing hate in OSN and mass-marketing fraud.