Sustainable Security: Exploring Longevity Challenges and Solutions for IoT

by

Conner Bradley

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs

in partial fulfillment of the requirements for the degree of

Master of Computer Science

Carleton University

Ottawa, Ontario

Abstract

The Internet of Things (IoT) has become increasingly integrated with our everyday lives providing physical and direct value to societies across the world. While small embedded devices are increasingly becoming integrated into products by IoT device vendors, so do our concerns about the longevity of these integrations. Unlike general-purpose computers, IoT devices are expected to be in service for long periods of time. While an IoT device may only need to perform simple tasks over its lifespan, the surrounding networked environment and potential threats will evolve. To ensure that IoT devices remain secure over extended periods and are not compromised by adversaries, they need to be supported throughout their entire lifespan. This places a significant burden on device vendors who are reluctant, and sometimes technically unable to maintain software for decades after deployment.

In this thesis, we examine IoT device longevity through the lens of security. Specifically, we are interested in the aspects of IoT device security that limits device longevity. To begin, we focus on understanding the current landscape of software updates in IoT. To our knowledge, software update practices in current IoT devices

are not yet well understood, despite a large body of research aiming to create new methodologies for keeping IoT devices up to date.

We then discuss a major shortcoming of current software update systems for IoT, which is characterized by a single point of failure: the IoT device vendor. Without support and updates from the device vendor, the software for the IoT device will become outdated and may experience negative consequences due to the constantly evolving security landscape. To overcome this challenge, we propose a new vendor agility approach for IoT device longevity. This approach would allow devices to receive support from sources beyond their first-party vendors.

Finally, we implement part of our vendor agility model to demonstrate feasibility on embedded devices. Our aim is to demonstrate how cross-platform embedded code can be created without the need for proprietary tools. Our proof of concept serves as a starting point for future research and work to break the dependency between devices and vendors, enabling long-term support and security for embedded systems.

Acknowledgements

First and foremost, I would like to thank my supervisor Dr. David Barrera for his guidance and support throughout my time at the Carleton Internet Security Lab. His mentorship has been instrumental in shaping my research and guiding me through the intricacies of my academic journey.

I would also like to thank the members of my committee, Dr. David Knox and Dr. Lianying Zhao for taking the time to read my thesis and provide their valuable thoughts and insights. Their feedback has significantly enhanced the quality of this thesis.

I would also like to thank my colleagues in the Carleton Security Research Labs (CCSL and CISL) for their support and the many stimulating discussions, which have been a constant source of inspiration and motivation. I would also like to extend my appreciation to CCSL's dilapidated server infrastructure, which, in its own unique way, provided an unexpected but much-needed distraction from my research.

Finally, I would like to thank my friends and family for their support and encouragement – this milestone would not have been possible without your support.

Contents

List of Tables

List of Figures

List of Code Listings

Chapter 1.

Introduction

Longevity is a problem that impacts all software developers, whether they consciously consider it when creating software or not. Once a piece of software is created from source code to executable, or from script to interpreter, programs themselves remain static and unable to change or evolve on their own over time. However, the surrounding environment is subject to change and evolution. New protocols emerge, rendering existing ones obsolete. Cryptographic algorithms become outdated, and as hardware advances, the system interfaces that developers rely upon also evolve. This poses a challenge for software that depends on services and functionality from interconnected systems, as sudden changes or non-backward-compatible updates in those systems can cause the software to break.

Not all software requires adaptability or change. Systems that don't require external connectivity, such as embedded devices running firmware without internet access,

have traditionally been deployed without any mechanism for software updates. Since they don't rely on internet connectivity, the complexity of their firmware is significantly reduced, allowing for simpler software designs that can be thoroughly verified.

However, when embedded devices require internet connectivity, the complexity of the device software increases dramatically. The additional layers of abstraction for network stacks, encryption, and wireless radio drivers contribute to a much more complex system. What was once a device with a relatively small set of verifiable requirements now becomes a system with heightened complexity and numerous unknowns that cannot be fully addressed during the creation of the device's firmware. These unknowns involve future changes that impact assumptions and designs made in the past such as encryption sustainability, core protocol changes, and the possible impact of changes from external dependencies such as Network Time Protocol (NTP) [85].

The IoT has grown rapidly in recent years, resulting in the widespread use of internet-connected devices in our daily lives. These devices are typically built using small, inexpensive embedded systems that run embedded firmware. One key feature of IoT devices is their integration into appliances, vehicles, and other everyday objects. As a result, they often have a much longer expected service life compared to general-purpose computers. While a typical computer may last for 5-10 years before being replaced, IoT device hardware has the potential to endure for up to 30 years or more. However, the main limitation of IoT device lifetimes is not their hardware, but

their software, as outdated software can render the device obsolete and vulnerable to security threats [125, 57].

The rationale behind the extended lifespan of IoT devices can be attributed to the lifespan of their analog counterparts [49]. Consider a conventional light switch, these can remain functional for several decades before requiring replacement due to wear and tear. The lifespan of this purely analog device is the benchmark for its IoT counterpart: an IoT-enabled light switch should be able to meet, or exceed, this benchmark in terms of service life.

This longevity of IoT devices introduces unique challenges. Firstly, the hardware and firmware of these devices must be designed with long-term reliability in mind. Ensuring that the embedded systems can withstand the test of time becomes crucial. Moreover, software updates and security patches need to be regularly maintained to keep these devices secure and up to date, even over an extended lifespan.

1.1. Motivation

A concerning trend in the IoT industry is increasingly smaller device lifetimes [57, 92]. From the perspective of hardware durability, IoT devices could very well meet the benchmarks set by their analog counterparts despite being more complex [49]. The main problem harming IoT device lifespans is not the hardware. Rather, it is the software that runs these embedded devices.

As we previously mentioned, interconnected systems greatly increase the complexity of the software needed to run them. The additional functionality and complexity of IoT device software implies some form of ongoing maintenance or support from the original device vendor. Generally speaking, the original device vendor is the only entity that is capable of supporting these IoT devices, as they possess the tooling, methods, and knowledge of the proprietary embedded systems that they create.

When a vendor ceases to support an IoT device, the device will no longer receive software updates, thus becoming stagnant and failing to keep up with the evolving technological landscape. While the world around it adapts and strengthens itself against emerging security threats, the stagnant device remains vulnerable. This lack of support poses risks not only to the owners of these devices but also to the entire IoT ecosystem. In the early days of IoT, this issue was not a significant concern for device users. However, consumers are now increasingly aware of the limited support provided by device vendors, typically lasting only 2 to 5 years. This limited support renders devices that could potentially last for decades practically useless [57].

The resource constraints of IoT devices present a challenge when considering long-term software deployment models. Many of the strategies that have been successful in other industries cannot be directly applied to IoT due to resource constraints. Ensuring decades of backward compatibility and support – such as the application compatibility shims used by Microsoft Windows and the Windows NT (NT) kernel – involves additional complexity and expensive abstractions that are not feasible

for resource-constrained embedded devices [34, 35]. IoT devices often operate with limited processing power, memory, and storage capacity, making it difficult to accommodate the necessary software updates and maintenance required for long-term support.

1.2. Problem

We are particularly interested in what the security research community can do to enable longer lifespans for IoT devices. Thus, finding new hardware-based solutions to this problem is out of our immediate scope. Specifically, we are interested in how the industry is approaching keeping IoT devices functional and secure for long periods. Within this context, functionality and security are simultaneously required for IoT devices to achieve longevity. If an IoT device loses functionality and the manufacturer is not providing support, the end user will replace it. Likewise, if an IoT device is not secure due to a lack of support, the increased vulnerability of the device will negatively impact devices and potentially create harm to broader IoT ecosystems [10].

We seek to understand the current landscape of firmware updates in IoT. We want to understand – beyond the vacuum of IoT security research – in the real world, are IoT devices being updated regularly by their vendors? As previously mentioned, a lack of software updates for IoT devices harms both the security and functionality

of devices. Furthermore, what levels of standardization and while there are several novel designs for secure firmware update systems oriented towards embedded devices, are any of these designs being used? Likewise, if vendors are not adopting solutions from the security research community, are they using solutions that are comparable in security measures?

Once we have our understanding of the current landscape of maintenance and support from vendors to deployed IoT devices, we are interested in what measures can be taken to extend support periods. Specifically, a common theme with IoT software update systems is an inherent dependency between IoT devices and vendors. For example, if vendor x manufactures and distributes a given IoT device, only that vendor is going to have the ability to develop and distribute software for that device. This inherently creates a single point of failure; without vendor x in existence, the devices it manufactured and supported will no longer receive software updates, preventing even another vendor or open-source community y from supporting the now unsupported devices from vendor x.

This particular edge case is commonly overlooked in current IoT software update systems, few consider the possibility of long-term support beyond a single vendor's existence. We believe the solution to this new paradigm involves decoupling the IoT device from its vendor to enable vendor agility.

1.3. Contributions

The contributions in this thesis are focused on improving the longevity and security of IoT devices, and the main contributions are divided into three parts. The first contribution **C1** involves an assessment of the current security landscape of software update systems in IoT devices. The results of this assessment serve as motivation for the subsequent contributions **C2** and **C3**, which propose a new security paradigm based on longevity and outline the implementation of a solution aimed at enhancing both IoT device longevity and security. We expand on the details of these contributions below.

C1 Identification and evaluation of firmware updates. In Chapter 3 we conducted an empirical study to examine the security and overall landscape of firmware updates in consumer IoT devices. As far as we know, this study represents one of the first comprehensive analyses of consumer IoT network traffic specifically focused on identifying communications related to software updates. During our investigation, we discovered common design patterns employed by several IoT devices and identified potential vulnerabilities that could be exploited. We present a detailed case study of various software update schemes and practices that we uncovered through our methodology. Furthermore, we provide an event-based characterization of IoT device update behavior. We offer insights into the different conditions that prompt IoT devices to initiate updates. Overall,

our study sheds light on the security aspects of firmware updates in consumer IoT devices, presenting real-world examples and vulnerabilities. By conducting this analysis, we aim to contribute to the understanding of the current state of firmware updates in the consumer IoT landscape, highlighting areas that require attention and improvement to ensure the security and integrity of these devices.

C2 IoT device longevity as a new security paradigm. In Chapter 4 we show that longevity and durability are not commonly applied perspectives when it comes to building software and security measures for IoT software. Given that many IoT devices are being deployed in situations where a long-term deployment model[1] is applicable, we believe that these devices should have additional measures to ensure that a device can continue to function, even after the original device creator stops supporting it or ceases to exist. We pose new problems of longevity and durability in IoT software to combat the growing e-waste crisis and ensure functional devices do not end up in landfills. We argue that the lack of long-term support for IoT software leads to device vulnerabilities and loss of functionality over time. To overcome this challenge, we propose a new approach that assumes the original manufacturer may no longer support the device, necessitating alternative methods for secure and continuous updates.

[1]Long-term relative to other devices. General-purpose computers may have a deployment model of 5-10 years, while IoT devices may be expected to last for 20-50 years.

Drawing parallels with other industries, we present a blueprint for designing IoT software/firmware stacks that explicitly account for the potential abandonment of the first-party vendor. We advocate for a decentralized approach where trusted parties take over security updates and patches. We introduce heuristics for devices to autonomously determine when vendor support ends and transition to a community-supported update channel. Finally, we address the challenges of securing IoT software transitions with a comparison between centralized, distributed, and hybrid approaches for ensuring the longevity of software updates.

C3 Proof of concept implementation of IoT vendor agility. In Chapter 5 we take the aforementioned model as a blueprint and provide a proof of concept implementation of our envisioned vendor agility system for IoT devices. We implement a platform-independent runtime designed specifically for the resource constraints of IoT devices. Our proof-of-concept WebAssembly runtime is designed to run on bare metal, or in conjunction with an operating system, to provide the basis for how IoT software developers from different organizations can provide support using the aforementioned vendor transition model.

1.4. Related Publications

The majority of the content presented in Chapters 3 and 4 has been published as a peer-reviewed publication and accepted for publication, respectively. Both publications were written primarily by the author of this thesis, Conner Bradley under the supervision of his supervisor Dr. David Barrera, who played a crucial role in supervising the publications.

- C. Bradley and D. Barrera. Towards characterizing IoT software update practices. In *Foundations and Practice of Security*, pages 406–422. Springer, 2023. DOI: 10.1007/978-3-031-30122-3_25
- C. Bradley and D. Barrera. Escaping Vendor Mortality: A New Paradigm for Extending IoT Device Longevity. To appear in *proceedings of the 2023 New Security Paradigms Workshop*, NSPW '23, Segovia, Spain. Association for Computing Machinery, 2023

Chapter 2.

Background

This chapter provides the needed background information to contextualize discussions and ideas in later chapters. Section 2.1 provides a thorough definition of what the Internet of Things means in the context of this work. Section 2.2 provides an overview of operating system designs in IoT with a comparison to general-purpose computing to further develop a picture of the limitations of IoT devices. In Section 2.3 we provide an introduction to longevity and durability in the context of IoT software, which is applied throughout Chapters 3 and 4. We introduce software update schemes for IoT in Section 2.4, with an analysis of long-term longevity in these update schemes. Finally, Section 2.5 provides some context on WebAssembly that is relevant to Chapter 5.

2.1. The Internet of Things

The Internet of Things (IoT) is a broad term that describes objects (things) that are capable of sensing, actuating, and communicating information between themselves. There is an inherent cyber-physical nature for IoT devices, as the sensing from sensors and peripherals and actuating in response to sensory input allows these devices to perceive and act upon the physical world. Generally speaking, IoT devices include any physical device with some form of network connectivity – not necessarily internet connectivity – that has sensors and/or actuators that enable the extension of an interconnected system into the physical world. The "things" in IoT can be existing objects or devices such as home appliances, vehicles, and industrial systems, or these things can be completely new devices that have not previously been possible.

At this point, readers would expect a more technical definition of what an IoT device is. What characteristics, traits, and unique values set an IoT device apart from any other computer? This is the problem with IoT: because of the increasingly broad umbrella of what is encompassed by the Internet of things, it is impossible to define what IoT is and what IoT is not. For example, the "I" in IoT refers to the internet, yet IoT devices do not need to connect to the internet [45, 19, 21], nor do they need to use the commonplace protocols that are used on the internet (e.g., TCP/IP). As a whole, IoT has become a misnomer; however, IoT remains the term of choice for researchers and industry.

It is crucial to define what subset IoT devices we are referring to as a target of this thesis. Our scope of IoT devices is based partially on resource constraints, but also on the intended use case of the device itself. Resource constraints are only partially useful as we discuss in Section 2.1.1, while there are ways of quantifying what is and is not an IoT device based on the resource-constraints of a particular device, the broader research community is largely in disagreement of existing quantification of IoT.

2.1.1. Resource-Constrained Devices

The Internet of Things is largely made up of resource-constrained devices. IoT devices are purpose-made with a particular goal or application known when a product is being designed, which allows product engineers to decide on the appropriate hardware for the particular application of an IoT device. Ideally, the hardware chosen by engineers is minimal enough for the device's intended usage. While this process optimizes the product's hardware, the converse effect of this is that minimal hardware imposes resource constraints on those building software for IoT devices.

Resource constraints on IoT devices are typically classified based on the specifications of the device's hardware [90]. Such specifications include the processing capabilities of the Central Processing Unit (CPU), Random Access Memory (RAM) capacity, and nonvolatile storage capacity. The IETF has a standardized set of device classes for resource-constrained devices, we summarize the classes presented in

Class	RAM Size	Flash Size	Example Platforms	Software
Class 0	<<10 KiB	<<100 KiB	ATmega48, PIC 16F	Application-specific firmware
Class 1	∼10 KiB	∼100 KiB	STM32F1, LPC1100	Specialized OS, firmware
Class 2	∼50 KiB	∼250 KiB	nRF52832	Specialized IoT OS, firmware
Class 3	>250 KiB	<1 MiB	ESP32, nRF52840	Specialized IoT OS, some general-purpose OS

Table 2.1.: The table of device classes presented in RFC 7228 [25] has been augmented to include a newly added Class 3. While Class 3 is not part of RFC 7228 [25], it becomes necessary to consider new device classes as IoT devices become more powerful, with increased RAM and flash storage, in order to keep up with industry advancements.

Request for Comments (RFC) 7228 in Table 2.1 along with some example IoT development platforms and use-cases that would fit these resource constraints [25]. Note that RFC 7228 was originally written in 2014 – 9 years ago at the time of writing – and the development platforms for IoT along with the resource capabilities of these devices have evolved since then. We present a new class, Class 3, which encompasses the more powerful and modern microcontrollers and SoCs commonly used in resource-constrained device development.

The addition or modification of constrained device classes is a common theme in many other works. Bellman et al. suggest a Class 2+ of devices, which shares some motivational similarities to our proposal of a Class [22]. There have been efforts to

propose new classes to RFC 7228, a working draft by Bormann et al. proposes a Class 3 of IoT devices which include 100 KiB of RAM and ∼500 KiB - ∼ 1000 Kib of flash, in addition to proposing a Class 4, Class 10, among others [26]. Similarly, El Jaouhari et al. proposes classes 3, 4, and 5 with similarly increased specifications for these higher classes [45]. Overall, the sheer amount of new device classifications points towards an aforementioned theme in IoT: the research community seems largely in disagreement on what an IoT device is, thus any quantified measurement or "binning" of IoT devices by hardware class is largely a poor measure of what an IoT device is and is not.

As shown in Table 2.1, the resource constraints of IoT devices typically preclude them from using general-purpose operating systems as the hardware itself does not have the needed features and capabilities to support the extra overhead of general-purpose operating systems. Class 0 IoT devices typically use custom-made firmware (discussed further in Section 2.2.4) instead of an operating system. Devices in classes 1 and 2 have more flash and RAM, allowing them to use IoT-specific operating systems instead of purpose-made firmware.

Another key difference that has large implications for building secure software on IoT devices is most microcontrollers lack a Memory Management Unit (MMU), instead having a Memory Protection Unit (MPU) or no form of hardware-assisted memory protection, which we discuss further in Section 2.2.

2.2. Operating Systems for IoT Devices

Most general-purpose computers rely on the abstraction of a process, which is an instance of a program managed by an operating system. This abstraction of a process relies on certain hardware features such as privilege levels, MMU, and large amounts of RAM allowing general-purpose operating systems to provide this key abstraction for application developers [108].

Resource-constrained microcontrollers used in IoT may only have a small subset of the features needed to use a general-purpose operating system. In this section, we review the limitations of embedded devices and discuss embedded operating systems designed to deal with these limitations.

2.2.1. Protecting Memory

One of the major hurdles in building operating systems for embedded devices is providing clean and secure abstractions of system memory. In conventional general-purpose Operating System (OS), this abstraction is typically taken care of by virtual memory, which presents programs with an abstraction of physical memory. Instead of programs using the physical addresses of the memory they occupy, the operating system will map physical memory addresses to virtual ones that are used by the program. This facade allows the OS to present memory that may be fragmented across many physical regions, or even paged to disk [108].

Mapping virtual memory is typically assisted by a hardware component called a MMU, which translates virtual addresses to physical addresses, providing memory protection, and enabling memory sharing among multiple processes. The MMU achieves these tasks by implementing techniques such as paging or segmentation [108].

In the context of the microcontrollers and SoCs used in IoT (see Table 2.1), a MMU is not typically found on resource-constrained IoT devices. Thus, operating systems designed to run on these resource-constrained devices cannot leverage virtualized memory. Instead, embedded devices employ a Memory Protection Unit (MPU). The primary purpose of an MPU is to provide memory protection and access control on a per-region basis. Regions of memory can be marked as readable, writable, or executable, to provide a coarse-grained isolation mechanism. Note that this means that software on these devices also cannot leverage virtual memory, thus, all the memory addresses that are available to software are the physical memory addresses, not a mapping to a physical memory address.

2.2.2. Monolithic Kernel

In the context of an operating system's kernel, a monolithic kernel is a type of operating system kernel architecture where all operating system services and functionality share the same address space. The monolithic nature of this type of kernel refers to all kernel functionality running in privilege ring 0. In a monolithic kernel, all services that are technically separate operate without any enforced separation [108, 105].

A monolithic kernel includes core system functionality such as process management, memory management, device drivers, file system support, and network protocols. These components are tightly integrated and operate in privileged mode – occupying a single address space, and thus a single privileged domain – allowing them direct access to hardware and system resources.

While monolithic kernels are inherently less secure due to no privilege separation between internal kernel components, monolithic kernels tend to outperform their microkernel counterparts due to there being no overhead for switching between address spaces.

2.2.3. Microkernel

Microkernels, in contrast to monolithic kernels, aim to provide the least amount of functionality in ring 0 and instead abstract kernel functionality into isolated services running in user space. Microkernels divide system services into separate, isolated modules called servers. These servers communicate with each other through well-defined message-passing mechanisms, such as Inter-Process Communication (IPC). Microkernels are more modular than monolithic kernels as individual kernel services can be updated independently, not requiring a single monolith to be re-built [73].

While the extreme approach taken by microkernels provides excellent isolation between components, the primary drawback of a microkernel approach is the increased overhead of communication between servers. This additional overhead is one large

reason many general-purpose operating systems today are not microkernel-based. In embedded contexts, the additional overhead poses additional challenges for Real-Time Operating Systems (RTOSs) [105].

2.2.4. Firmware

Firmware is a specialized type of software that is embedded within hardware, typically stored in Read-Only Memory (ROM) or Programmable Read-Only Memory (PROM) [104]. Firmware is specifically designed to run on particular hardware and is designed to perform a single, specific function. This function can vary widely depending on the device it is associated with. For example, firmware in a Solid State Drive (SSD) will be responsible for wear-leveling flash, handling requests to the flash storage, and other tasks that are specific to a SSD. Because of the limited scope of what a device's firmware is responsible for, firmware can be developed without the additional complexity and resource requirements of an operating system [105].

In contrast, a kernel within a general-purpose OS provides abstractions and facilities to enable the broad usage of hardware. It serves as the core component of an OS, responsible for managing system resources, scheduling tasks, and facilitating communication between software and hardware. Unlike firmware, a kernel is designed to support a wide range of applications and functions, providing a platform for processes to run and interact with the hardware more flexibly.

2.3. Longevity and Durability of Software

One of our primary objectives is to increase the longevity of IoT devices. However, to work towards this goal we must first outline what longevity means. Longevity describes the period a given product is useful (i.e., its useful "life"), where the period is measured from when a product is placed into operation to the moment it is decommissioned or discarded [84]. Longevity is thus a subjective measure of how long a product can remain useful to its environment [39].

Within the context of computer science, the longevity of software can be thought of as the relevance of the functionality it provides at the current point in time. For example, if an IoT operating system only supports Bluetooth Low Energy (BLE) networks, it may have poor longevity if its current environment deprecates BLE in favor of a different incompatible network.

While *longevity* is an entirely subjective measure of how useful a product is within a specific context, *durability* describes objective measures of lifespan. Durability is the period when a given product was designed to be functional [37]. Factors driving durability are primarily objective, such as the type and quality of materials used, the maximum lifetime of components as listed on data sheets, and the expected usage of the device that can cause wear and tear [49]. Factors impacting durability also impact reliability, these factors are generally correlated.

Unlike a physical tangible product, the durability of software is harder to quantify.

Software durability can be thought of as software quality, where a highly durable piece of software should accomplish all of its projected requirements with high levels of reliability, no bugs, and no vulnerabilities. Within the context of software durability, any bug or vulnerability[1] will negatively harm the durability of the piece of software.

We continue this discussion in Chapter 4, where we further expand on the significance of longevity and durability of IoT device software and analyze potential solutions for extending the lifetime of IoT devices.

2.3.1. Software Updates

As we discussed in Chapter 1, programs are static. Unless the program is self-modifying – which is a very uncommon practice – a program cannot change or evolve itself. In the bigger picture of longevity and durability, the longevity of software can be negatively impacted by the static nature of programs. For example, protocols, APIs, and interfaces that a piece of software depends on may evolve, and this evolution may lead to breaking changes that cause a loss of functionality. This loss of functionality impacts longevity, as a less functional device may be making the device less useful in its broader environment.

The continuous evolution of software environments also impacts durability. As critical protocols evolve to adapt against new threats or pre-existing bugs, a static

[1] Assuming the software is intended to be free of bugs and vulnerability. For example, if vulnerability is an intentional requirement then a vulnerable piece of software could still be considered highly durable as it would be functioning as intended.

piece of software will not inherently receive the benefits of the evolving environment. For example, if a vulnerability was found in a shared library responsible for ensuring the confidentiality and integrity of transmitted data – such as OpenSSL – the static nature of a program means that even if the vulnerability is fixed, it will not reap these benefits.

Given the long-term deployments that several types of IoT devices may face, software update systems for IoT should be hardened to withstand long-term issues. Issues we have seen impact IoT devices over long periods include outdated and vulnerable cryptographic implementations, which can impact the security of encrypted communication channels [29, 91].

2.3.2. Software Update Schemes for IoT

Prior surveys of software update systems for IoT by El Jaouhari et al. [45] and Bellissimo et al. [20] comprehensively overview various traits of software update systems for IoT devices, however, these prior surveys do not adequately consider the longevity of the update schemes themselves.

We chose a selection of commonly-researched software update schemes for IoT devices, and other software update schemes that have been applied in some capacity in a IoT context. For example, if an update framework was originally designed for non-IoT contexts but a future work augments or enhances the framework in an IoT context, we include it below. We are particularly interested in common frameworks

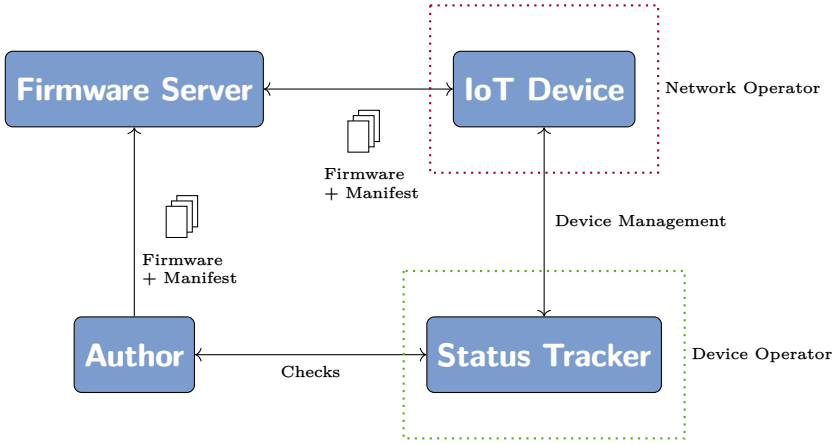


Figure 2.1.: Overview of the architecture of IETF SUIT [81]. The device operator is responsible for the day-to-day operation of a fleet of IoT devices, and the network operator is responsible for the network that the IoT device connects to.

or frameworks that meet some of our criteria for longevity, defined in Section 2.4.1.

The Internet Engineering Task Force (IETF) created the SUIT project to create a standards-track software update system that is generic and flexible enough to encompass several IoT deployment verticals [81, 45, 56]. SUIT defines an architecture for generating and distributing firmware to IoT devices. SUIT defines a manifest model that contains metadata to describe the firmware image, such as dependencies, cryptographic information to validate the image, and information on the version and author of the firmware.

Figure 2.1 shows the overall architecture of IETF SUIT. An author who creates and manages firmware for IoT devices can distribute the firmware and associated mani-

fest to a distribution server. IoT devices managed by a network operator can fetch firmware and manifests from the distribution server, with device state information being synchronized to a status tracker which is managed by the device operator.

The Update Framework (TUF) by Samuel et al. is a novel software update scheme that protects developers against compromise of signing keys [101]. TUF makes use of several protection mechanisms such as utilizing multiple key roles to limit damage if a single key is compromised. The framework utilizes timestamping and snapshotting to ensure metadata remains valid for a specific time and creates a consistent view of the repository, guarding against tampering. By treating each update as a separate release with a version number, TUF ensures that only legitimate updates are accepted. Furthermore, TUF supports delegated roles, enabling the distribution of responsibilities and signing authority among entities, which mitigates the risk associated with individual key compromise. Finally, the framework allows for offline revocation mechanisms, enabling the invalidation of compromised keys without an online connection.

Uptane by Karthik et al. is based on TUF with modifications that make it suitable for automotive applications [64]. Uptane's threat model goes beyond securing against attacks during the update process and also considers the entire supply chain security. Modifications and enhancements to TUF include the addition of a director repository, which allows a software vendor to have more control of software images deployed in individual automotive Engine Control Units (ECUs).

Asokan et al. designed an architecture for Secure Software Updates of Realistic Embedded Devices (ASSURED) [13]. ASSURED is designed to include all stakeholders in an IoT ecosystem – namely the device vendor, firmware distributor, domain controller, and IoT device – to provide end-to-end security and attestation between all connected entities. The architecture is based on enhancements of TUF, and is currently the most IoT-applicable variant of TUF.

Lightweight M2M (LwM2M) is not a standalone software update system, rather, it is a protocol suite for IoT devices that encompasses most IoT device operations and lifecycle management, which includes software updates [74]. LwM2M is designed for low-power resource-constrained devices, primarily using Constrained Application Protocol (CoAP) as an application protocol and Datagram Transport Layer Security (DTLS) to ensure message integrity, authenticity, and confidentiality.

UpKit by Langiu et al. is a lightweight software update framework for resource-constrained IoT devices [68]. UpKit aims to encompass firmware generation, propagation, verification, and installation. Upkit's architecture uses a double-signature process to ensure firmware freshness to prevent rollback attacks. An additional verification step allows for rejecting invalid firmware early in the update process, allowing the device's bootloader to reject invalid firmware images before any installation occurs.

Baton by Barrera et al. was not originally designed for resource-constrained IoT devices, rather, it was designed as an extension to Android's app installation framework

to allow for certificate agility [17]. Through Baton's certificate agility mechanism, developers can delegate the signing authority of an application to effectively transfer applications to a new maintainer, or update existing singing keys with new ones.

There are more comprehensive surveys of software update schemes for IoT that compare broader sets of criteria that may also impact longevity [45, 19, 56]. Our focus for this section is not to present a survey of every IoT update scheme in existence, rather we are interested in commonplace software update schemes intended to address the challenges of IoT deployments, and additionally, other non-IoT software schemes that meet some of our criteria of longevity, which we discuss further in Section 2.4.1.

2.4. Longevity of Software Update Schemes

With the importance of software update schemes established for IoT devices, this section compares existing software update schemes for IoT through the lens of longevity. Note that some of the update schemes we have chosen were not inherently designed for IoT, but they were originally designed for forms of IoT devices[2].

To aid in the comparison of existing update schemes, we present criteria for evaluating the longevity of firmware update schemes by looking at the longevity of the cryptographic implementation used, and by looking at any measures that enable the update scheme to switch between vendors. This includes evaluating the considera-

[2]Depending on what an "IoT device" is defined as in the context of the work, see Section 2.1 for more information on how we define IoT device.

tion of long-term cryptography, along with measures that allow the update scheme to switch between vendors. We then apply the criteria in Table 2.2 for a full comparison of update schemes against these criteria for longevity.

2.4.1. Criteria for Longevity of Software Update Schemes

With common software update schemes established, we will now introduce our criteria for the longevity of software update systems. In all the aforementioned update schemes, cryptographic algorithms are used at various stages to ensure that the distribution of the firmware is confidential, that the firmware is not tampered with, and that its integrity is upheld during transport. Additionally, cryptographic measures are used to provide protection against several other potential attacks such as rollback attacks, fast-forward attacks, and mix-and-match attacks, among many other tactics that can be employed by adversarial parties [101, 20].

Over time, these cryptographic measures can become increasingly weak due to evolving attacks. As a result, existing cryptographic algorithms are updated to keep them safe from evolving attacks. If a cryptographic algorithm is found to be vulnerable, a system should be able to easily and readily switch to a different algorithm. This concept is referred to as cryptographic agility and cryptographic agility is becoming increasingly more important for long-term deployments in the post-quantum era [16].

We also focus on the inherent single point of failure that all of these update schemes have in common: the creator of the software. The software creator is usually the

vendor who manufactures a given device, if they cannot provide support for any reason, then a device will no longer receive software updates. The implications and analysis of this problem are discussed further in Chapter 4.

The themes of ensuring cryptographic and vendor longevity are important, however, they are not the only major themes of longevity. The criteria that we present below are non-exhaustive; however, this initial outline of agility criteria serves as a starting point for focusing on the gap in this research area.

Specification of cryptographic implementation: Does the specification or proposal leave the cryptographic algorithm choices up to the vendor as an implementation detail, or is there specific advice on which cryptographic algorithms to use? A ● indicates that there are algorithms specified, and a ○ indicates this is up to the adopter of the update scheme.

Cryptographic Agility: Does the specification or proposal consider the implications of cryptographic agility? Over time, vulnerabilities and exploits will be developed for leading cryptographic algorithms which further weaken them; cryptographic implementations and algorithms need to be updated to remain secure. Thus, if an update scheme is using cryptographic algorithms to ensure confidentiality, authenticity, or integrity, of updates, then a lack of consideration of this over long-term periods may further weaken the update scheme. A ● indicates consideration of cryptographic agility, a ◐ indicates mention of cryptographic agility, and a ○ indicates no consideration of the problem whatsoever.

Vendor Delegation: Can the specification or proposal handle multiple entities that support a single IoT device? Specifically, if an update scheme has any particular entity as the only entity that can distribute firmware for a device, this creates a single point of failure that cannot be handled by the update scheme itself. An ● indicates that the specification can handle multiple entities redundantly supporting a single device, alleviating single points of failures on a single vendor. A ◐ indicates that the update scheme supports multiple entities building components but a single entity is still responsible for distributing firmware, and a ○ indicates no forms of vendor delegation are considered, thereby indicating the scheme is single-vendor only.

Vendor Agility: Vendor agility can be thought of as a type of vendor delegation, where no entity manages the delegation. In other words, full vendor agility implies that the IoT device can autonomously select a vendor that is actively supporting it in a decentralized manner, whereas vendor delegation may be accomplished through a centralized system. A ● indicates that vendor agility is supported, and a ○ indicates that there is no consideration of vendor agility.

Update Scheme	Specification of cryptographic implementation	Cryptographic Agility	Vendor Delegation	Vendor Agility
SUIT [81]	◐	◐	◐	○
TUF [101]	○	○	○	○
LWM2M [74]	◐	○	○	○
ASSURED [13]	○	○	●	○
UpKit [68]	●	○	○	○
Baton [17]	●	○	◐	○

Table 2.2.: Comparison of software update for IoT devices, we compare our preliminary criteria for longevity of a software update system. Criteria involving cryptography analyze the overall security of a software update system over long periods of time, and criteria involving the software vendor analyze the potential point of failure of a single vendor system.

2.4.2. Longevity Comparison of Software Update Schemes

We present our comparison of the longevity of software update schemes in Section 2.3.2 against our criteria from Section 2.4.1 in Table 2.2. In Table 2.2 we list SUIT as one of the most complete update schemes for longevity as it meets (albeit partially) most of our criteria. However, there is still room for SUIT to improve. Listed below are the shortcomings of SUIT relative to our criteria:

- For specifying a cryptographic implementation, the main RFC associated with SUIT[3] does not specify cryptographic algorithms or criteria for selecting cryptographic algorithms, aside from noting that the system integrators should choose them carefully [81]. However, a relatively new draft for a follow-up RFC does have clear requirements for what cryptographic algorithms should be used in specific use cases [79]. The draft is incomplete and is also a working document, thus we consider this a partial effort for meeting the criteria.

- For consideration of the implications of long-term cryptography, IETF SUIT also only partially has a solution for this. IETF SUIT mentions that algorithms should be chosen carefully and that one of the worst-case estimates for quantum-accelerated key extraction is 2030 [81, 62]. However, SUIT does not have any required measures for cryptographic agility, which is thought to be one of the main ways of ensuring that legacy cryptographic algorithms get phased out as

[3]RFC9019

they become weaker and more vulnerable.

- For vendor delegation, the general architecture of IETF SUIT allows authors from different entities to distribute firmware to a firmware server. This means that a vendor can hypothetically allow other vendors to develop and push firmware updates for IoT devices, somewhat alleviating a single point of failure from relying upon a single vendor.

- For vendor agility, IETF SUIT does not allow devices to autonomously change firmware servers. Thus, nothing is met here.

The Update Framework (TUF) does not fully meet our criteria, mainly because it wasn't explicitly designed with long-term viability in mind [101].

On the other hand, Lightweight M2M (LWM2M) presents a more comprehensive set of protocols, encompassing not only IoT software updates but also various other aspects of IoT device lifecycle and functionality [74]. This makes it a valuable option for IoT ecosystems.

An interesting solution is the Architecture for Secure Software Update of Realistic Embedded Devices (ASSURED) by Asokan et al. which is built upon the foundations of TUF [101] and Uptane [64]. Designed for the automotive sector, it allows multiple stakeholders to collaboratively develop and distribute firmware, with a single separate entity handling firmware distribution [13]. This approach seems promising for the specific challenges faced in the automotive industry, as it allows for multiple

stakeholders to integrate intellectual property on a single ECU.

UpKit by Langiu et al. also shows promise, particularly in the area of vendor delegation, as it offloads the responsibility of firmware distribution to a separate entity. This alleviates a single point of failure from the device vendor but adds a new single point of failure to the firmware distributor [68].

As for Baton by Barrera et al., though initially intended for Android applications, it does possess a well-defined set of cryptographic algorithms. Furthermore, it can perform vendor delegation by using multiple copies of signatures for apps, allowing vendors to delegate responsibilities under specific circumstances. It's worth noting that this delegation capability was not an original design requirement for Baton [17].

Overall IETF SUIT is the most complete and would be an attractive candidate for expansion. We discuss how update schemes for IoT can be modified and extended to meet our criteria for longevity in Chapter 4.

To summarize: the problems of longevity in IoT devices we discussed in Chapter 1 can be tackled using continuous software updates to ensure the IoT devices can keep up with the evolving technological landscape it is surrounded by. However, when we look at software update schemes for IoT devices (and even software update schemes beyond IoT) we find factors of longevity are rarely considered. The criteria we established are high-level and non-exhaustive but serve as a starting point to embark on improving the longevity of IoT device ecosystems.

While our analysis covers schemes that are analyzed by the broader IoT research

community, this does not mean that these schemes are being used in practice. To further our understanding of software update systems in IoT, we delve into the current landscape in Chapter 3. We aim to gain valuable insight into the state of software update systems in real-world IoT devices.

Our analysis also does not cover any solutions for the problems of longevity in IoT devices. It is largely understandable that of the update schemes that consider what happens when a vendor needs to delegate authority it is marked as out of scope of the update framework itself. In Chapter 4, we dive deeper into the problem of longevity and attempt to design a high-level architecture that can solve the challenges of longevity that we previously discussed.

2.5. WebAssembly

In Chapter 5 we implement a platform-independent runtime for IoT devices to demonstrate the feasibility of the vendor agility model presented in Chapter 4. Specifically, we implement a WebAssembly runtime. In this section, we present relevant background information for that chapter.

All WebAssembly binaries take the form of a module, which acts as a top-level container for everything contained within it [98, 53]. Modules contain a list of sections that declare definitions functions, memories, tables, global variables, and static data. These definitions can then be exported for use externally, and similarly, definitions

can be imported from outside sources.

For a WebAssembly runtime to execute a module, it must be instantiated. The process of instantiating a module involves providing definitions for all imports, validating the WebAssembly module, and creating the state of the WebAssembly Virtual Machine (VM) based on what is declared in the module. Code inside a WebAssembly module is organized as a vector of functions [98]. Functions can be called by other functions within the module or can be exported to be called externally.

To enable efficient execution, WebAssembly employs a stack-based virtual machine (VM) model. This means that computations are performed by manipulating values on a stack rather than using registers. WebAssembly instructions operate on these values, performing operations such as arithmetic, control flow, and memory accesses.

WebAssembly also provides a linear memory model, allowing modules to allocate and access a contiguous block of memory. This memory can be used to store data and facilitate interactions between WebAssembly and the host environment. The module can define memories and specify their initial and maximum sizes.

In addition to functions and memories, WebAssembly modules can declare tables, which are essentially arrays of references. Tables enable efficient indirect function calls, where the target function is determined dynamically at runtime. By using tables, WebAssembly can provide support for dynamic dispatch and function pointers.

WebAssembly modules can also define global variables, which are mutable values shared across functions within the module. Global variables can store various types

```rust
fn cube(n: i32) -> i32 {
  n.pow(3)
}
```

```
(func $cube (param i32) (
   result i32)
    local.get 0
    local.get 0
    i32.mul
    local.get 0
    i32.mul)
```

Figure 2.2.: Example Rust function (left) with WebAssembly text representation (right). The toy example of a cube function on the left simply raises its argument `n` to a power of 3. The equivalent WebAssembly function accomplishes this by multiplying the argument by itself times.

of data, such as integers, floats, or references. However, it's important to note that global variables should be used sparingly, as their misuse can lead to performance issues and/or non-deterministic behavior.

To facilitate communication between WebAssembly modules and the outside world, the module can define imports and exports. Imports are declarations of entities (functions, memories, tables, or globals) that the module requires from the hosting environment. Exports, on the other hand, make selected entities within the module accessible to the hosting environment. This mechanism enables WebAssembly modules to interact with the surrounding ecosystem, making them versatile building blocks for various applications.

WebAssembly modules can be created from several commonplace languages, such as Rust, C, and C++, among many others. The common thread between most languages

that compile to WebAssembly is they are strongly typed[4], which is a requirement for a WebAssembly compiler as it must be able to emit the correct types when creating WebAssembly. In Figure 2.2 we show an example function in Rust alongside the WebAssembly emitted in WebAssembly Text format (WAT). Note that WebAssembly is a binary format, and WAT is equivalent to the textual format of the binary, similar to how assembly language is the textual format to machine code instructions.

[4]Some projects compile weakly-typed languages, such as JavaScript, to WebAssembly. Such projects make use of extensive type annotations in source code and compile-time checks to enable the possibility of compiling to WebAssembly.

Chapter 3.

Software Updates in IoT: An Empirical Study

Software updates are critical for ensuring systems remain free of bugs and vulnerabilities while they are in service. While Internet of Things (IoT) devices are capable of outlasting desktops and mobile phones, their software update practices are not yet well understood, despite a large body of research aiming to create new methodologies for keeping IoT devices up to date. This chapter examines efforts towards characterizing the IoT software update landscape through network-level analysis of IoT device traffic. Our results suggest that vendors do not currently follow security best practices, and that software update standards, while available, are not being deployed.

3.1. Introduction

Consumer Internet of Things (IoT) devices have gained significant popularity in recent years, resulting in a revolution of IoT devices used in many applications. IoT devices are typically resource-constrained and require specialized operating systems and software stacks depending on their application [21]. Due to the unique resource constraints of IoT devices, device vendors have to either design their software update infrastructure and supporting applications from scratch or use an integrated third-party solution[1] which has historically shown to be inconsistent and vulnerable [122]. Software update systems are well understood and widely available on general-purpose computers and servers [20]; however, there is little insight and research into how these vendor-specific IoT software update systems work due to a lack of standardization in the IoT space [123, 23]. The goal of this chapter is to characterize how typical consumer IoT devices query for and retrieve software updates, and evaluate the security of these techniques as used by prominent IoT vendors.

As discussed in Chapter 1 a unique challenge for deployed IoT devices is their expected lifespan. Typical personal computers have a relatively short lifespan compared to an IoT device, which is expected to behave in an appliance-like fashion with minimal (if any) downtime. Personal computers may get replaced in 5-10 years if the hardware cannot keep up with current software demands. In contrast, an IoT

[1]Such as Microsoft Azure IoT, or Amazon Web Services IoT.

device such as a smart thermostat may be expected to run for decades before being replaced. With the constant evolution of technology, device vendors have the additional challenge of providing a secure implementation of their software on legacy devices.

We hypothesize that suboptimal update intervals from IoT device vendors may further weaken IoT update systems. For example, device libraries such as the crucial OpenSSL library were analyzed during a study of 122 IoT device firmware files, which revealed several vendors failed to patch OpenSSL in their IoT devices after critical vulnerabilities were released [125]. Device vendors took months to supply an updated system image with a patched OpenSSL version, and one vendor took nearly 1,500 days to patch the critical vulnerability. Failing to update critical libraries causes these devices to gain a larger attack surface that could potentially be leveraged by bad actors to trick the device into downloading malware [117] or to bypass security measures that are in place to prevent the device from loading modified firmware [40, 23].

In recent years, there have been many proposals for secure software update systems that are designed for IoT [27, 55, 125] and related cyber-physical systems [64, 86]; however, there is no research (to our knowledge) aiming to broadly understand the IoT software/firmware update landscape in consumer IoT devices.

Our primary focus is identifying software updates being requested and taking place. The benefits of this can be leveraged in various contexts: Network-level update de-

tection can be used as independent feedback to end users that their devices are being updated regularly – an IoT device vendor may promise to publish security patches for their IoT devices, but not deliver on that promise [125]. In an enterprise context, administrators may want to apply the principle of least privilege to fleets of IoT devices. Certain IoT devices do not need continuous access to the open internet as most devices can function exclusively with Local Area Network (LAN) connectivity to a central hub or other devices. The only edge case to this is checking for updates and downloading them. If an active firewall can detect update-related traffic from IoT devices, it can adjust rules to (1) allow the IoT device to download an update from the internet, and (2) log the update instance.

The research contributions in this chapter are:

- The first in-depth analysis of consumer IoT network traffic to identify software update communications. We identified design patterns used in several IoT devices and found vulnerabilities that could be exploited.

- A case study of software update schemes and practices that we identified through our methodology. Devices featured in our case study distribute software updates over Hypertext Transfer Protocol (HTTP) with no tamper-resistant protection mechanisms added on. One of the devices identified in the case study provides a happy medium between update transparency and security.

- An event-based characterization of when IoT devices update. We contextualize

the various conditions that lead to an IoT device performing updates. For example, power cycling an IoT on is highly likely to trigger an update check.

3.2. Methodology

Our research objective is to understand and characterize how and when IoT devices perform software updates. To accomplish this, we build a network traffic analysis system that identifies and analyzes software update requests and responses from IoT devices. We aim for the system to be vendor-agnostic, requiring no *a priori* knowledge about the IoT vendor's infrastructure or devices. The system should also identify updates across multiple independent cloud vendors, which are relied upon heavily in IoT.

To accomplish this, we analyze network traffic from a 2019 Internet Measurements Conference (IMC) paper by Ren et al. [97] which actively captured traffic from 81 IoT devices. These 81 devices were located in two geographic regions; 46 in the United States (US), 35 in the United Kingdom (UK), and 26 common devices across both regions. In total, the dataset contains packet captures from 55 unique devices. Collected data was harvested at network gateways, but no form of middle-person attack was done on TLS traffic which precludes peering into an encrypted device communication. Therefore, in this paper, we rely exclusively on extractable HTTP

traffic for identifying software updates[2]. Additionally, we harvest metadata from the TLS handshakes to gain insight into the security of the secure communication channels used by these devices.

3.2.1. Data Extraction

In total, the dataset of packet captures from Ren et al. is 13 Gigabyte (GB) in size, which includes 37,744 packet captures recorded by the automated test system and 611 unsupervised experiment packet captures, yielding a total of 38,355 packet captures. We do not separate traffic by geographic region as Ren et al. found negligible differences in region-specific traffic [97].

To identify network traffic related to software updates, we hypothesize that update interactions between an IoT device and vendor cloud follow a structured schema. If the schema is human-readable (e.g., JavaScript Object Notation (JSON), Extensible Markup Language (XML), etc.) there will be keywords contained inside indicating some update-related information, such as a firmware version. We initially searched for a single keyword "update", which led us to build a corpus of update-related keywords: update, upgrade, firmware, software, and download.

These keywords will be the basis we use for identifying update-related traffic; however, manually searching through files will not scale to the number of devices we have.

[2]Note that this shortcoming inherently limits our observations of specific update protocols to devices that do not implement measures to secure communications; our observations may only be representative of a lower bound of devices.

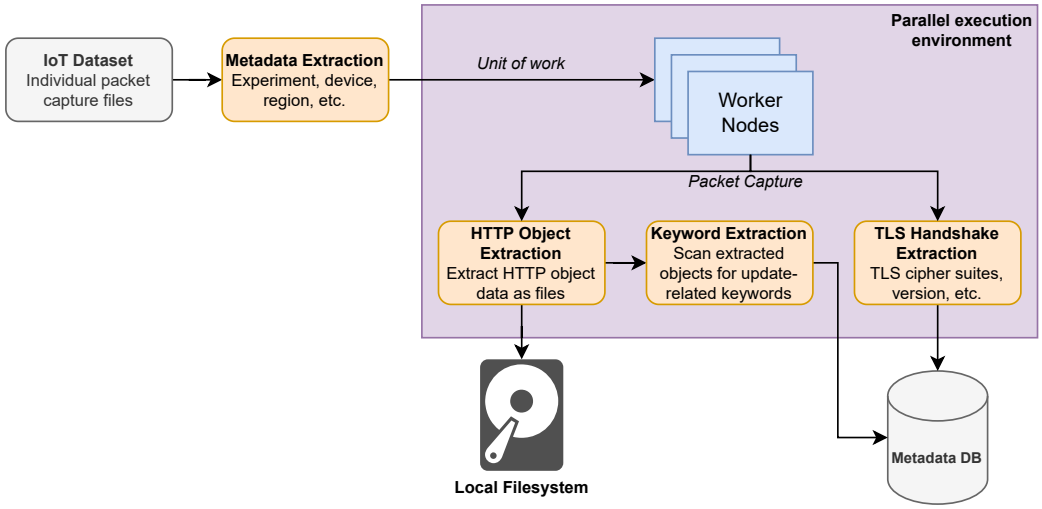


Figure 3.1.: Data extraction pipeline: starting with the IoT packet capture dataset, we extract metadata for each packet capture to represent a given packet capture as a unit of work. We process each packet capture in parallel, extracting HTTP objects to the local filesystem, TLS Handshakes, and update-related metadata. All extracted metadata is stored relationally in a metadata database for further analysis, and HTTP objects are stored on the local filesystem.

Therefore, we developed a parallel network traffic processing pipeline (see Figure 3.1) that manages network traffic metadata and HTTP object extraction. The pipeline design is compatible with distributed data processing frameworks such as Apache Spark, and works on the dataset as follows:

Metadata Extraction: We extract metadata representing the packet capture. This includes the specific sub-dataset, region, experiment type (e.g., power on, interact with the device, etc), and device name. The extracted metadata is saved to a metadata database and used for later steps in the pipeline.

Parallelization: We parallelize the extraction of metadata and HTTP objects on a per-packet capture basis. The parallelization is done by assigning each packet capture to a worker node, and the worker node performs the following steps on each packet capture individually. In practice a parallelization approach is not needed; however, passive analysis of a large amount of packet captures warrants the speedup gains of parallelization.

HTTP Object Extraction: We extract all HTTP payloads from a given packet capture. The HTTP payload data is of particular interest as it provides us insight into any files transferred along with any web service interactions.

TLS Handshake Extraction: We then extract TLS client and server hello data using a modified version of `pyshark`[3]. Our modified version of `pyshark` supports extracting an extended set of TLS handshake metadata, including the ciphers adver-

[3]https://github.com/KimiNewt/pyshark

tised in the TLS client hello and server hello handshake. In total, we return a list containing every TLS handshake, including the TLS version, TLS handshake type, and a list of cipher suites. The TLS cipher suite data is used to determine if devices are adequately securing communication channels against TLS-related attacks.

Keyword Extraction: For each of the extracted HTTP objects, we scan for the aforementioned update-related keywords by performing a case-insensitive search for all of the keywords. A keyword occurrence flags a packet capture related to a software update. Counts of keyword occurrences are saved to the metadata Database (DB) for future analysis.

The data-extraction pipeline operates per packet capture in parallel. On a test VM with 24 virtual processors, 64 GB of RAM, and a solid-state drive, we were able to run the extraction pipeline on $38,355$ packet captures in over 60 minutes, with approximately 10 packet captures processed per second. Without a parallel approach, our extraction pipeline would have taken over 24 hours to complete.

3.2.2. Data Analysis

Using the metadata that corresponds to the packet capture, we can perform extended analysis on the packet capture that had been flagged as having update-related traffic. After identification of these packet captures, we inspect the HTTP response data to look for any update endpoints or update artifacts. Ideally, we should find no update-related artifacts in HTTP responses, as this would imply these files are transmitted

over an insecure channel.

Device vendors *should* be protecting their firmware from being tampered with regardless of the transfer protocol being used: if a vendor uses only TLS to secure their updates in transit, the compromise of a single cryptographic key is the only requirement to jeopardize the integrity of the vendor's update system [101].

Analyzing IoT update interactions by raw traffic can be misleading as it does not consider the *context* that triggers a device to update, only that the device checked for an update. To further characterize update interaction, we look at event-related information to provide more context to the various conditions that cause IoT devices to update. All the packets captured from the Ren et al. study are labeled with various event-related information such as power events, app interaction, or idle events. Therefore, we analyze these crucial pieces of context to correlate events to update activity. For example, if an IoT device checks for an update when powered on, an adaptive firewall can use temporal data of an IoT device's network connectivity to provide more context to classify if an IoT device may be requesting and applying a software update.

Finally, we extract and analyze all TLS handshake data from all the packet captures (independent of update keyword traffic) to assess the overall strength of the communication channels in use. Our methodology only allows us to perform extended analysis on unencrypted traffic; however, if IoT devices send all of their traffic over an encrypted medium, it is a reasonable assumption that the devices will also

perform firmware updates over these encrypted connections. If the TLS implementation on the IoT device is outdated or insecure, this will undermine the overall security of the IoT device, including the software update system. Whether TLS is explicitly or implicitly chosen for a design, using TLS is a design choice for IoT update systems. Considering the historical vulnerabilities associated with TLS, the use of outdated TLS versions can negatively harm device security and longevity [66].

To interpret the set of cipher suites advertised between clients and servers, we converted the cipher suite's hexadecimal value to the Internet Assigned Numbers Authority (IANA) cipher suite name by leveraging a cipher suite information Application Programming Interface (API) [99] which aggregates all IANA cipher suites along with IANA cipher suite security classifications. Cipher suites are then categorized into four buckets: insecure, weak, secure, and recommended. Insecure cipher suites have easily exploitable security flaws and thus should *never* be used, while weak cipher suites may have proof-of-concept vulnerabilities that are more difficult to exploit in practice. The classes of secure and recommended cipher suites have no known vulnerabilities, and all recommended cipher suites are a subset of secure cipher suites. The only differentiating factor is that recommended cipher suites support Perfect Forward Secrecy (Perfect Forward Secrecy (PFS)).

3.3. Results

In this section, we discuss our results in identifying update-related traffic. At the network level, software updates are difficult to detect if the update communications are taking place over an encrypted connection. TLS offloading may be an option in non-IoT contexts; however, attempting TLS offloading on IoT devices will require physically tampering with the device which may cause erratic behavior [97].

Our HTTP object extraction pipeline extracted HTTP objects from 5,766 of 38,356 packet captures, which is 15% of the packet captures in the dataset. In other words, 85% of packet captures use some form of encryption, or a protocol other than HTTP. We extracted HTTP data for 35 out of 55 devices[4], which is 63% of devices. Originally, Ren et al. attempted to measure encryption adoption with slightly different results: no device had more than 75% unencrypted traffic [97]. The key difference in our results is we focus on extractable HTTP objects, whereas Ren et al. attempted to guess if certain User Datagram Protocol (UDP) traffic was encrypted or not by measuring byte entropy, which only concludes if certain packets are *likely* encrypted [97].

In the following sections, we describe our results for identifying software update keywords, characterizing software updates based on device interaction, and our TLS results. These results are summarized as follows:

- 3.3.1: Out of the 35 devices that did not encrypt all traffic, 9 (25%) checked for

[4]Originally, Ren et al. had 81 devices with 26 common devices between regions, thus 55 unique devices.

available software updates transparently.

- 3.3.2: IoT devices check for updates during power and idle events, and a small percentage of devices check periodically (some as often as once per hour).
- 3.3.3: Update endpoints (where software update files are hosted) for devices in our set exist primarily in 3rd party cloud service platforms, or on content delivery networks (CDNs), which makes Domain Name System (DNS)-based identification difficult.
- 3.3.4: TLS is pervasively used in IoT communications, possibly including update-related traffic. Devices that only use TLS for communication could be vulnerable to key compromise if there are no additional protections in place [101].
- 3.3.4: The majority of our devices use secure TLS cipher suites which would not make them vulnerable to TLS downgrade attacks; however, there are devices that support vulnerable TLS cipher suites, which jeopardizes any update communications made through TLS.

3.3.1. Update Keywords Results

We successfully extracted several HTTP interactions between IoT devices and web services related to software updates. Our most prominent keyword is *update* with 1,351 occurrences among extracted HTTP objects, *firmware* with 639 occurrences,

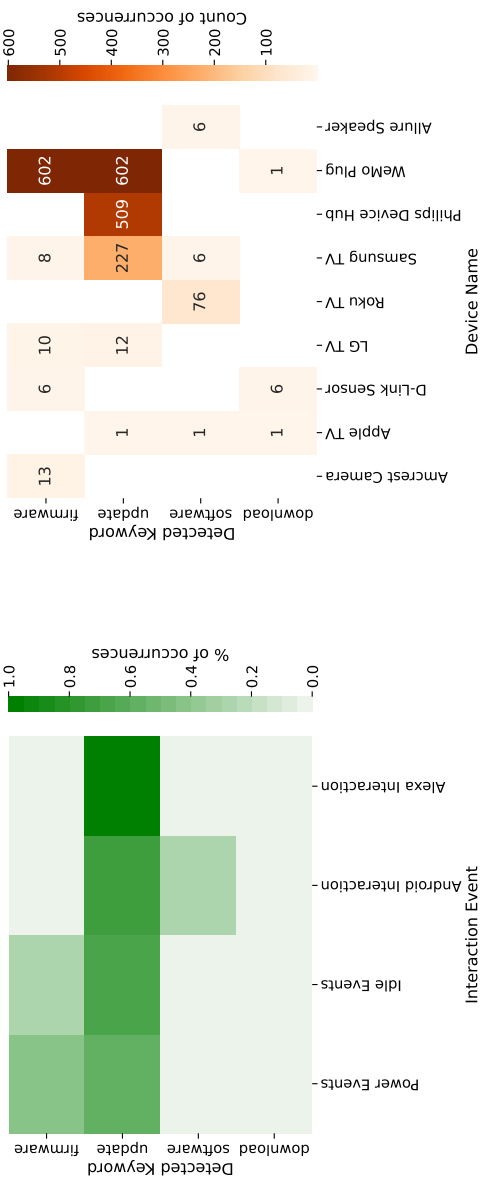
software with 89, and *download* appearing only 8 times.

The specific devices and the corresponding keywords they matched are shown in Figure 3.2b. The heatmap shows the number of occurrences of the keywords in the rows for the devices in the columns, where a darker blue indicates more occurrences. We observed that certain devices exchange update-related information much more often than others, such as the Wemo plug and Phillips hub.

The Wemo plug device had the most occurrences of keywords, which means the Wemo plug was polling the most frequently for updates; however, this does not imply there may be a software update in progress. For example, the Wemo Plug exchanges firmware information in nearly every request which contributes to the high amount of keyword detection; however, we did not find any proof that the Wemo plug performed an update during the capture period. There is an update web service offered by the Wemo plug, which we discuss in detail in Section 3.4.3. By contrast, the Apple Television (TV) only has a single occurrence of exchanging update-related keywords, and we found that the Apple TV downloaded system firmware over HTTP, which would imply that the Apple TV installed the aforementioned firmware, which we discuss in Section 3.4.2. This contrast shows that our heuristic does not guarantee a device is performing an update, but it is enough to detect traffic that *might* be update related.

Aside from being able to detect firmware downloads in real-time, an unexpected result from our heuristic was it picks up current updates and firmware versions in

7 of the 9 devices. This is because these 7 devices report their firmware version as an HTTP request, or as part of a service discovery response. This is valuable information for both defensive and offensive applications. A potential application for this in defensive security is an active firewall appliance that can scan IoT devices and fetch firmware versions from them, if a Common Vulnerabilities and Exposures (CVE) is released for that particular firmware the firewall can automatically quarantine the affected devices. This assumes that the firmware version is accurately reported, which may not be the case for malicious devices. For offensive security applications, an attacker could perform reconnaissance by identifying vulnerable firmware versions of devices that actively advertise these versions.



(a) Update keyword occurrences by interaction event. A darker color indicates a higher percentage of occurrences. In the original study [97] there were 9 different Alexa interaction events, and 4 android interaction events, which we chose to merge into a single group for readability.

(b) Count of detected update keywords aggregated by device as a heatmap, where the number in the square corresponds to the number of keyword usage occurrences were found.

Figure 3.2.: Our results for update keywords by device and interaction event.

3.3.2. Update Events Results

Our results for event-related update activity are shown in Figure 3.2a. The heatmap shows the number of update keyword occurrences in the rows for the interaction event in the columns, where a darker color indicates more occurrences. Due to the granularity of the experiments from Ren et al., Android-related events (e.g., taking a photo, controlling a device from an app, etc.) and Alexa interactions (e.g., invoking Alexa, changing color, etc.) were merged into two respective categories. Aside from these events, all 9 of the IoT devices in Figure 3.2b exchange update-related keywords on power events, and even more on idle events. Examples of update traffic events include devices reporting their *firmware* version to an update service, then receiving an *update* response in return.

When idle, we found some IoT devices that exchange update-related traffic between one another. This is out of the ordinary, as independent IoT devices should not be issuing or exchanging update commands to one another when idle – these communications should only occur between the device and the vendor's update platform. We investigated these inter-device occurrences and found that as part of service discovery protocols (e.g., Simple Service Discovery Protocol (SSDP), UPnP) there is an exchange of firmware information. Certain devices even advertise endpoints for invoking update behavior manually which is ripe for exploit by bad actors or rogue IoT devices. Refer to the WeMo case study (Section 3.4.3) for more information regarding these endpoints.

Other than power and idle events, Alexa interaction events contribute the most to our heatmap. Alexa devices do not exchange detectable update-related traffic; however, the Philips hub exchanged update-related information when being controlled by Alexa. Additionally, the Roku TV, Samsung TV, and Wemo plug exchanged update-related data when controlled remotely by Android interaction events. We believe there is no correlation between these interactions and update traffic: these devices exchange the same information when not being controlled by Alexa or Android.

3.3.3. Observed Update Design Patterns

We analyzed the extracted HTTP interactions flagged as being update-related to attempt characterizing common designs or behaviors between device vendors. Unfortunately, we observed no common architecture or strategy was used between the 9 devices we identified. This can largely be attributed to the lack of vendor coordination in the IoT space, where vendors create their own update schemes rather than use (or adapt) solutions from industry standards or other vendors. The heterogeneity of the designs and schemas involved provide great motivation for standardized update system designs, such as RFC 9019 and RFC 9124 [81, 80]. While there is no common schema among different device vendors, we noticed some common patterns among certain device manufacturers.

No Security : The D-Link movement sensor, Amcrest camera, and Wemo fetch firmware update metadata from a web service that returns a complete Uniform Re-

source Locator (URL) for downloading the firmware image. What is concerning about this is there is no tamper-protection in place for any of these devices. To make matters worse, both of these devices fetch data from public S3 bucket endpoints over HTTP. We examined firmware images served through these endpoints and found no forms of tamper-protection such as checksums, digital signatures, or authentication built into the firmware.

Out-of-band Security : While insecure device update schemes are certainly concerning, there are update techniques that allow authentication and integrity verification even over HTTP. The Apple TV exchanged all update-related traffic over HTTP, including web service interactions for downloading the firmware and related metadata. What sets the Apple TV apart is it exchanges digital signatures and certificates over HTTP to validate the responses. Apple's design provides a happy medium of ensuring the integrity (assuming the signatures and certificates are validated) of the update through cryptographic means while giving us insight into specific details that can be leveraged by a network appliance, such as specific firmware and information assuming that the network appliance can parse the XML schema Apple uses.

Full TLS : The remaining devices encrypted all cloud-destined communications using TLS. It is reasonable to expect that, if implemented, a software update mechanism would also use one of the available TLS channels. While communication encryption is advantageous for security and privacy, we believe transparency in software update implementations (perhaps implemented with an out-of-band scheme as described

above) can be beneficial for providing transparency and security, as we described in Section 3.1. Additionally, we note that exclusive reliance on TLS for software updates is known to be insufficient in protecting against many update-specific attacks [101].

3.3.4. Cipher Suite Results

We see a larger amount of devices with extractable TLS cipher suites, which is expected as many IoT devices use TLS as a means of interacting with the web services they depend on. In Figure 3.3 we observe there were a total of 16 insecure cipher suites used between IoT devices. All 16 cipher suites have significant vulnerabilities that when combined with a downgrade attack could allow an attacker to perform a Machine-in-the-Middle (MITM) attack; however, among the 24 devices that advertise insecure cipher suites, we estimate 4 of them would be vulnerable to a downgrade attack. This is because the *secure* and *recommended* cipher suites would take precedence over the weak and insecure cipher suites, and the cipher suites contained in secure and recommended classes contain measures to prevent downgrade attacks.

We have only discussed the TLS cipher suites in the context of IoT devices. To see these results in perspective to other applications that require secure communication, we searched for a dataset of TLS cipher suite support in web browsers. While we did not find a comprehensive dataset that summarized recent browsers, we did find a service that provides us with what our browser supports [95]. Using this service, we found modern browsers (Firefox 94, Chromium 96) support far fewer cipher suites

with none of them being marked within the insecure category – although roughly half of the cipher suites supported were deemed to be "weak". This can offset the large number of IoT devices that offer "weak" cipher suites, which may be present solely for backward compatibility. Although these weak cipher suites in IoT devices do not significantly increase the attack surface compared to modern web browsers, they can still pose security risks when not using TLS 1.3.

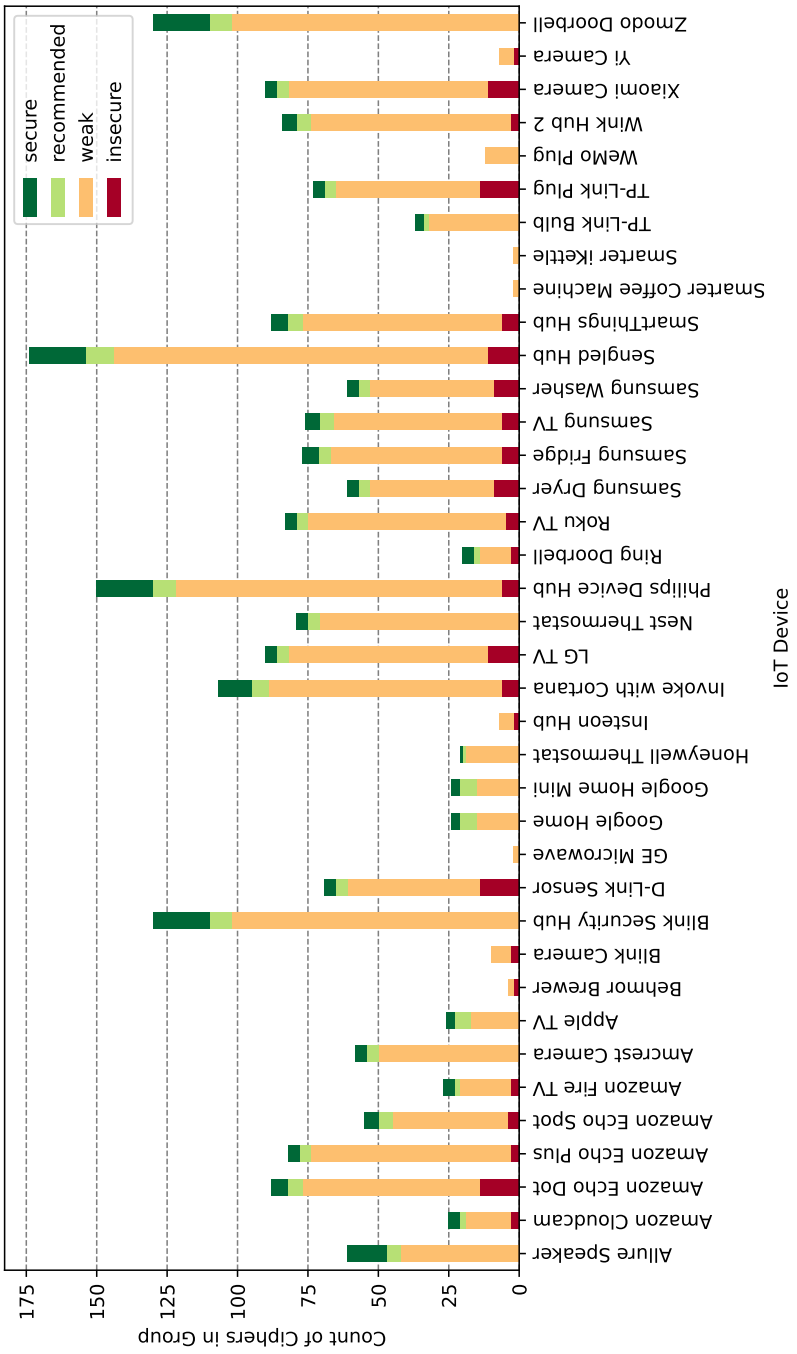


Figure 3.3.: Count of TLS cipher usage on a per-device basis. Each bar is represented by insecure, weak, secure, and recommended cipher suites.

3.3.5. Limitations

We found 11 devices that did not have extractable HTTP data or extractable TLS data. By manually inspecting packet captures we found several devices that stream data over UDP, which is a consistent finding with the Ren et al. study [97]. The data was not meaningful, as it was either encoded using some vendor-specific encoding or a stream of application-specific data (e.g., a video stream) that can not easily be deciphered. While these edge cases are technically possible to extract, it is challenging to do so at scale given the wide breadth of devices and a large amount of packet captures.

A limitation of our study is TLS encrypted traffic, which is consistent with other large scale IoT analysis papers [94, 91]. A potential workaround for TLS-encrypted edge cases is an alternative heuristic: for example, another approach that is agnostic to the protocol in use is to look at response sizes. If a device exchanges a large amount of data in a short burst, assuming that this burst of traffic is abnormal for the device based on regular behavior, is not ideal as there is no way to verify if traffic is update-related – this only identifies large bursts of abnormal traffic. Furthermore, even if we could deduce that encrypted traffic is a device update, there is no meaningful extractable information from an encrypted payload such as firmware version which is crucial to our motivation for detecting IoT software updates.

Additionally, our results derived from HTTP traffic are not representative of all IoT devices. This is due to the majority of IoT devices using TLS, thus our results

and case study derived from non-TLS traffic is representative of devices that are not attempting to follow security best practices. As a result, the practices of the majority of IoT devices could be misrepresented due to the results obtained from the lower-security minority of devices.

Another potential heuristic is to analyze traffic patterns temporally. O'Connor et al. developed a simple yet effective methodology for classifying various IoT subsystems without any form of decrypting or inspecting packet payloads, instead opting to analyze traffic frequency and size over a long period of time [89]. This temporal approach proved effective for identifying IoT device telemetry, and in an active measurement context, O'Connor et al. were able to derive various attacks based on a temporal analysis of IoT device traffic. While this approach is novel, it is not ideal for a large-scale passive analysis of traffic.

Regarding the keyword-based analysis, our heuristic which associates terms such as "firmware" and "software" to update-related events can produce false positives. For example, some devices report a current firmware version to a web service contained as an HTTP payload. While this is not an update request, our pipeline will flag it as such and require manual removal. Future work will investigate the use of additional heuristics to improve the accuracy of identification of updates without requiring manual verification. Adding checks for outbound traffic, inbound traffic, and schema verification would greatly assist in avoiding false positives.

3.4. Case Studies

In this section, we present two case studies detailing software update mechanisms used in the wild by three major vendors. First, we look at the firmware update interactions from the D-Link Camera, which we use to illustrate harmful practices that undermine the device's security. We then contrast this approach with the firmware update interactions we observed against the Apple TV, which combines several distinct tamper-resistant mechanisms with update transparency. Finally, we conclude our case studies with a vulnerable WeMo update service, that allows for unsigned code to be uploaded from an arbitrary source.

3.4.1. D-Link Camera Firmware

The D-Link camera is an example of the **No Security** pattern, as it exchanged firmware update information through HTTP. Based on the identified traffic, we extracted a firmware update endpoint and also a firmware image. The firmware update endpoint is a web service that accepts a device model and returns an XML response containing firmware metadata information along with a URL to the latest firmware download. We were able to download the latest firmware image as it is being hosted by a static file store which does not require any prior authorization. The firmware update endpoint does not return any checksum or signature to validate that the firmware

image was not tampered with. Using the `binwalk` utility[5] we analyzed the firmware image and found the following:

1. A μImage header, indicating that the OS is Linux built for a Microprocessor without Interlocked Pipeline Stages (MIPS) CPU. This is likely a boot loader for the next item

2. Lempel-Ziv-Markov chain Algorithm (LZMA) compressed data, likely the kernel image to be executed by (1)

3. A SquashFS filesystem, which is the root filesystem

The image header indicates that the OS is a Linux Kernel from roughly 2014 (9 years old at the time of writing). Looking at the kernel image (2) we extracted the image version, which is Kernel version `2.6.31` released in 2009 [110]. While we did not find any notable CVEs for this particular version (*2.6.31*) of the kernel [75], we did find CVEs for the parent minor version (*2.6*) which allow for arbitrary code execution through multiple buffer overflows [41]. It is likely after 2014 the device reached the end of its "service life", thus D-Link stopped updating it. This is unfortunately a fairly common occurrence amongst IoT devices [96].

Theoretically speaking, the D-Link camera is vulnerable to MITM attacks as shown in Figure 3.4: (1) the communication with the update service is unauthenticated

[5]https://github.com/ReFirmLabs/binwalk

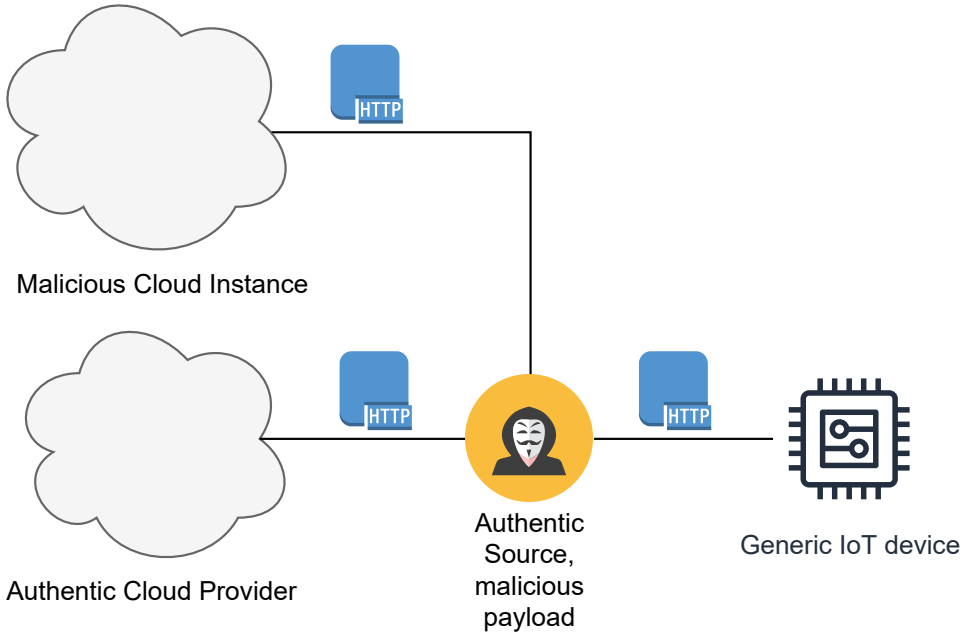


Figure 3.4.: An example MITM attack scenario that the D-Link camera is vulnerable to. The attacker would appear as an authentic source that provides a malicious payload, such as a download link to a modified firmware version being hosted by the attacker.

and does not have integrity protection; and (2) the communication with the image repository is unauthenticated and does not have integrity protection. For (1), an on-path attacker can intercept traffic between the IoT device and the vendor's cloud. In this case, the message responded by the vendor's cloud contains the full URL to the firmware image being hosted on an S3 bucket (also on HTTP). A second MITM attack (2) could occur if an attacker intercepts HTTP traffic between the IoT device and the S3 bucket. With this in mind, it is highly likely an attacker can leverage (1) to give the D-Link camera the URL of a different S3 bucket hosted on the "malicious cloud instance" which would then serve the modified firmware. An attacker could build and distribute modified firmware trivially, as the original firmware file is not signed, nor does it have any other protection mechanisms in place for the file.

3.4.2. Apple TV Firmware

In a contrast to the D-Link camera, the Apple TV combines transparency in updates with security. The complete update flow of the Apple TV is shown in Figure 3.5. Similar to the D-Link Camera update metadata is exchanged over HTTP; however, there are several additional measures to harden communications against attackers.

The Apple TV first connects to an updates repository over HTTP which returns an XML response containing available updates for that particular device, which is similar behavior to the D-Link camera. Although the connection for update metadata

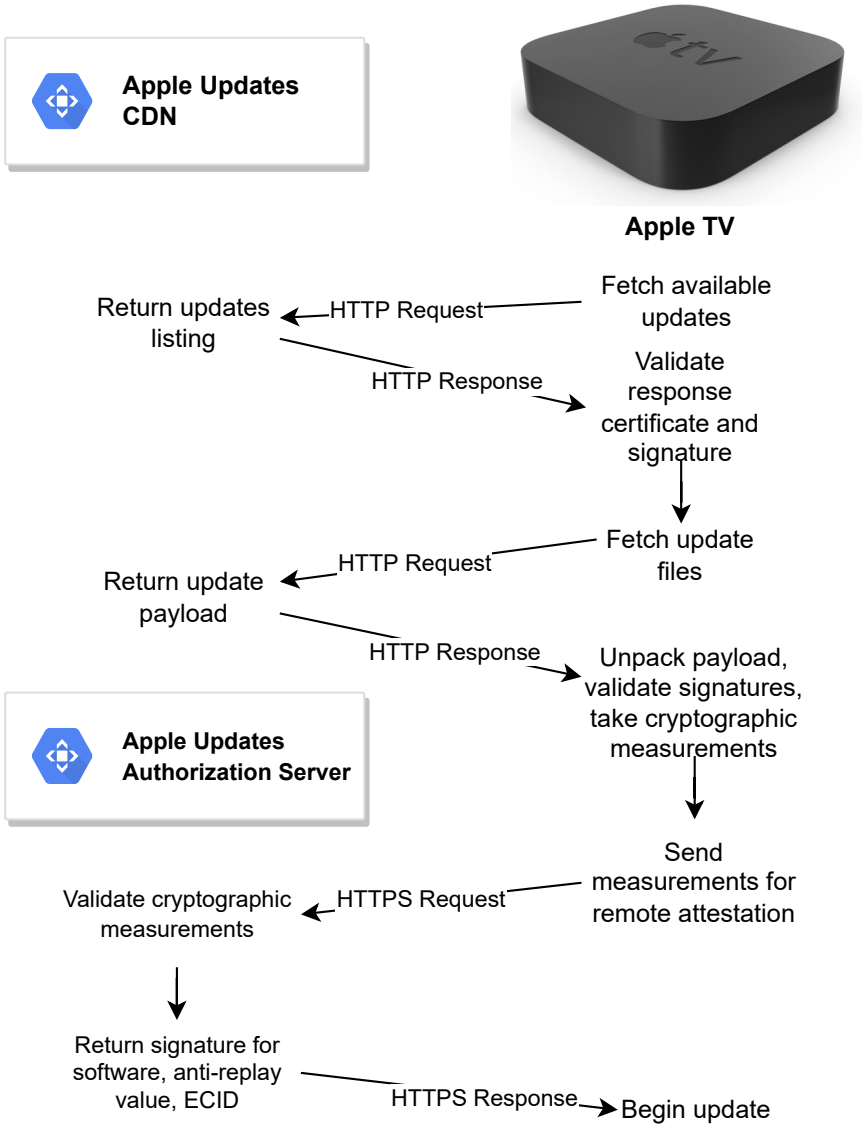


Figure 3.5.: AppleTV update process between Apple's content delivery network and update authorization server. There are two distinct stages to the update process: first downloading the update bundle via the Apple Updates CDN, then validating and authorizing the update through remote attestation.

happens over HTTP, we found the API response contains a certificate and signature field, which is used to validate the response [102]. Parsing the certificate using the `openssl` utility [2] we found the following: the certificate was issued by the "Apple iPhone Certification Authority", with a common name of "Asset Manifest Signing". This suggests that the certificate is purpose-made specifically for signing these update manifest responses. Additionally, the signature included in the response can be used to validate the integrity of the response. Unfortunately, the certificate expired in 2018, and the API response indicated updates from as recently as 2020.

When it comes to downloading the update, this communication also takes place over HTTP using a similar design to the D-Link camera. The Apple TV firmware repository contains a field that points to a content delivery network (CDN) that hosts the firmware image. One significant difference is there is a field that contains a measurement related to the update. Unfortunately, we could not identify how this measurement is derived; however, we are assuming that if the update file is downloaded and does not match the measurement, the update is invalid. This is consistent with Apple's platform security documentation which details the measures taken to secure device updates [102].

Using the Apple Repository response, we reconstructed the firmware download URL and acquired the firmware image for the AppleTV by downloading it over HTTP. The firmware image is distributed as a ZIP file, which when unpacked reveals a file tree for distributing software updates. Without having the source code to the software

responsible for performing updates on Apple devices, we are unable to determine how exactly the update is performed; however, combining an analysis of the directory tree with prior reverse engineering efforts [12] along with Apple's platform security documentation [102] gives us a relatively good understanding of how the update is performed.

After the AppleTV has validated the update payload, assuming the device-side verification of the update has no errors, the AppleTV must then perform remote attestation with the Apple Updates Authorization server to perform the update. According to our packet captures, this communication takes place over Hypertext Transfer Protocol Secure (HTTPS), so we do not have concrete knowledge of what exactly is being sent. According to Apple's platform security documentation, cryptographic measurements of the bootloader (iBoot), kernel, operating system image, and Electronic Control Identification (ECID) are sent to the update authorization server [102]. The server validates all the measurements sent by the device, and if they are valid, the update server returns the signature for the software, an anti-replay value, and the device's ECID [102].

3.4.3. WeMo Update Service

The Belkin WeMo plug largely communicates using Simple Service Discovery Protocol (SSDP), which is a protocol used to advertise services and consume them in

a standardized way [7]. SSDP uses HTTP as its underlying communication protocol, therefore all SSDP activity was captured by our passive analysis. We observed among the various device management services listed is one for firmware updates. The firmware update service advertised two actions: "GetFirmwareVersion" and "UpdateFirmware". The "GetFirmwareVersion" endpoint takes no arguments and returns a firmware version string. The "UpdateFirmware" endpoint takes several arguments including "NewFirmwareVersion", "ReleaseDate", "URL", "Signature", "DownloadStartTime", and "WithUnsignedImage". Particular arguments of interest to an attacker would be the "URL" and "WithUnsignedImage" fields, which indicate that the endpoint accepts arbitrary URLs along with being able to accept unsigned firmware images.

We, unfortunately, cannot test the viability of uploading arbitrary firmware to the WeMo device as we are only passively analyzing packet captures; however, our research into the aforementioned update service shows previous efforts have proven to be successful in exploiting the WeMo device[6] [32]. An attacker could have a local (or remote) firmware repository, and upload a modified firmware image to the WeMo device. The WeMo device would then attempt to download the modified firmware image and install it, similar to what we show in Figure 3.4. The only difference between the exploit used in the D-Link camera and the WeMo plug is the attacker

[6]Since the discovery of these vulnerabilities in 2013 WeMo has patched affected devices. Due to the vulnerability being remediated, we did not report our findings to WeMo.

has the ability to trigger device update behavior by interacting with an endpoint, whereas the D-Link camera has no such functionality.

3.5. Related Work

To our knowledge, this is the first work attempting to analyze and characterize how consumer IoT devices perform software updates at the network level. There have been recent works focusing on the different network-level analysis of IoT devices: Prakash et al. analyze the update practices of IoT vendors by tracking software versions listed in the user-agent header included in HTTP requests made by IoT devices [94]. The conclusions found by Parakash et al. do not characterize and analyze how IoT update systems work, rather, they conclude that IoT device vendors are slow to update their devices when new vulnerabilities are found.

We identified pervasive use of TLS, which precludes the identification of update-related traffic without additional data analysis. Related work here includes Alrawi et al., who provide an excellent Systematization of Knowledge (SoK) of the overall security of home IoT devices by systematizing the current state (as of 2019) of IoT vulnerability literature and then evaluating 45 devices, a subset of the security evaluation involves looking at various encryption qualities that would make the device vulnerable [9]. More recently in 2021, Paracha et al. performed a deep dive into IoT TLS usage patterns which ultimately found 11/32 IoT devices are vulnerable to in-

terception attacks [91]. If IoT devices are relying on TLS to secure communications to backend APIs and endpoints for software updates, any vulnerabilities in the TLS transport layer will undermine the overall soundness of how these devices perform updates.

An encouraging finding is the high amounts of TLS usage among devices; however, there is a caveat to this high TLS usage: it is only one line of defense. If a private key is compromised, this could jeopardize the integrity of update-related services if there are no additional lines of defense. Samuel et al. present a novel design for an updated system that allows for key compromise in update systems [101].

Due to the previously discussed challenges, there are several opportunities to explore and innovate IoT software update designs. Related work in this space consists of proposed designs for IoT update systems relating to firmware updates and library management. Zandberg et al. present a prototype for a firmware update system on IoT devices by leveraging various open-source libraries and standards [123]. Zandberg et al. leverage SUIT, a new IETF standard that provides *encrypted* firmware update files with encryption keys provided by hybrid public-key encryption [114]. The SUIT standard appears as if it may not work on resource-constrained IoT devices, but Zandberg et al. have their reference implementation built on IoT devices with less than 32 Kilobyte (KB) of RAM and 128 KB of storage [123].

3.6. Conclusion

Using a passive measurements approach and a dataset from one of the largest IoT information exposure studies to date [97] we identified and characterized several design patterns used by IoT devices to perform updates. There is no common schema or design pattern behind various update systems, which provides additional motivation for standardizing IoT software updates [81]. Additionally, we characterized events related to when an IoT device may update, which is useful for building data-driven models for real-time update identification. In our analysis of update systems, we found vulnerable devices that provide no mechanisms for securing firmware updates. We observed that many devices use encrypted connections to secure communications: 60% of devices support insecure TLS cipher suites, while 10% of devices in our dataset are vulnerable to downgrade attacks.

Our findings from this study paint a concerning picture regarding industry practices that impact the longevity of IoT devices. There are no discernible standards in use within the industry, and several instances of insecure cryptography, or even no cryptographic measures at all. Overall, this chapter has raised serious doubts about the long-lasting nature of current products and systems. The remaining chapters will delve into other potential barriers to longevity, as indicated by the identified shortcomings. The importance of software updates in the long-term maintenance and support of devices should not be underestimated. This chapter has shed light on

the existing opportunities for improvement in this crucial area.

Chapter 4.

Extending IoT Device Longevity

Internet of Things (IoT) devices are increasingly being treated as disposable, becoming unsupported shortly after deployment and ending up in landfills prematurely. IoT manufacturers lock devices to their ecosystems and prioritize the development of new devices over the support of legacy product lines. This chapter argues that a paradigm shift is needed to increase IoT device longevity. We review the unique challenges that IoT manufacturers face in extending device lifetimes, and identify software and security updates as a key requirement for device longevity. We propose a new IoT device software stack and lifecycle that allows devices to continue safe operation even after the vendor disappears. While we recognize that the sustainable design and management of IoT devices is a complex sociotechnical problem, we hope that the ideas in this chapter helps guide future discussions on this important topic.

4.1. Introduction

Research papers discussing the Internet of Things (IoT) often begin by citing IoT device deployment projections to highlight the current pervasiveness and growth trajectory of the industry. "double in size within the next four years" [69], "75 billion by 2030" [103], "a whooping trillion connected devices by year 2035" [65]. What these papers fail to discuss is the proportion of these IoT devices that will end up in landfills prematurely.

Our planet is facing a significant e-waste problem, exacerbated by the rapid end-of-life and deprecation of IoT devices [57, 58]. Consumers acquire these inexpensive devices and install them in their homes, only to discover that the device is only supported by the vendor for a short period (typically 1-2 years) [92, 36]. Once this support period is over, devices no longer receive feature updates and – more importantly – security updates. As a result, many of these devices are perceived as no longer useful or functional after a relatively short period of time, contributing to the growing global e-waste crisis [51].

Does this mean that IoT devices cannot be designed and built to last as long as their analog counterparts? In other words, is it even possible to design an IoT device that remains in operation and useful for over 25 years? A cursory look at the internals of some off-the-shelf IoT devices reveals that the *hardware* is generally up to the task of durability. Solid-state internal components are rated for several decades, with

flash storage being one of the main components that wear out over time, followed by mechanical actuators that enable the device to interact with the physical environment (e.g., relays, servos, etc.). Broadly speaking, engineering embedded hardware that lasts decades is feasible[1], but what about the software?

Writing long-lasting *software* for embedded devices is also feasible, and indeed the industry has been doing this for a long time. Embedded systems have been integrated in consumer appliances and electronics for several decades, well before the IoT revolution. Embedded systems have also been heavily utilized in various sectors, including automotive, manufacturing, and healthcare. The challenge arises when developers are tasked with writing long-lasting software for an embedded device that requires use of *the internet*. A substantial increase in complexity occurs when internet connectivity is brought into scope: evolving network protocols, bugs in cryptographic algorithm implementations, and changes to APIs of external services forces IoT devices to be updated [6], or stop working and get thrown out.

This (sometimes unintentional) programmed obsolescence in IoT software requires a new paradigm to solve. Users cannot be expected to keep track of which critical network libraries are no longer updated on their devices, and solder serial jumpers onto their devices to flash unofficial fixes. Vendors cannot be expected to maintain devices indefinitely as this fundamentally conflicts with their business models. We

[1] We recognize that engineering hardware designed to withstand harsh environmental conditions (e.g., weather, high-impact, military environments) is much more challenging. Our scope herein is focused on the gentler environment of a modern home.

cannot legislate permanently secure code into existence. Open-source software and hardware does not magically address this issue either; while access to the source code helps third-party maintainers write new code for abandoned devices, allowing a third party to overwrite the software on a device is indistinguishable from an attack [60].

In this chapter, we argue that the lack of long-term support for IoT software directly contributes to the global e-waste crisis and that a radically different approach is needed to keep functional IoT devices out of landfills. Our position is that if a device requires internet connectivity, it will inevitably become non-functional and/or vulnerable over time. While users may not immediately notice that their device is vulnerable, they are more likely to notice the disappearance of features and functionality. A unique challenge in achieving long-term updates is that the IoT device may outlast the manufacturer, and thus all technical means for designing, building, and distributing an update may be destroyed, or locked away behind intellectual property protections [87].

This chapter reviews the state-of-the-art in IoT software updates to highlight why these approaches are independently unsuitable for long-term device maintenance. We discuss why oft-cited alternative solutions (e.g., software updates as a paid service, device leasing, etc.) will also be insufficient moving forward. We draw parallels to the plastics and automotive industries to showcase differences and similarities across these vastly different domains.

With this context, our blueprint for long-term updates assumes that the first-party

vendor is a single point of failure, so we propose a new strategy for designing IoT software/firmware stacks that explicitly presumes the first party will abandon development. Then, security updates and patches can be taken over securely by another trusted party. This decentralized approach has worked in the personal computer domain, where systems may continue to work for decades as long as the architecture and device drivers are supported by the general-purpose operating system. We note, however, that we are not proposing a general-purpose OS for IoT devices, as IoT hardware is too heterogeneous. Instead, we propose a more explicit demarcation between firmware (code unique to the device, which may be proprietary in nature) – and software (code with no hardware-specific ties).

Next, we suggest a set of heuristics (e.g., heartbeat messages to supporting cloud infrastructure, timestamps of last software releases, etc.) that can be used for the IoT device to autonomously determine whether its vendor is no longer providing updates, and transition to a community supported update channel transparently, if it exists. We discuss the technical challenges in selecting and using these heuristics.

The final component in this new paradigm is securing the transition to the new update channel, potentially over several decades. During this time, cryptographic keys may be leaked or compromised due to insufficient bit length. Cryptographic protocols themselves may be found to be vulnerable. We discuss how the software update literature has largely ignored longevity factors, and we present an initial discussion of tradeoffs between centralized, distributed, and hybrid approaches for securing soft-

ware updates.

4.2. On IoT Software and Firmware Updates

IoT device manufacturers cannot predict the future: they do not know what the next several years (or even decades) of security vulnerabilities, bugs in existing code, and breaking API changes will hold for them. Therefore, IoT device firmware requires the ability to be updated such that manufacturers can fix, patch, or add new features to deployed devices. The absence of firmware updates leaves devices vulnerable to security threats and exploits [10, 117]. Hackers and malicious actors frequently target outdated and unsupported devices, as these devices are often more susceptible to attacks due to unpatched vulnerabilities. Unsupported devices may become part of botnets which can be used to launch large-scale cyberattacks [10, 117].

In the context of IoT devices, the term *firmware* refers to the operating system image that contains the kernel, libraries, and applications. On resource-constrained IoT devices[2], the firmware is typically monolithic (sometimes referred to as a unikernel), consisting of a single binary that provides all hardware abstractions and application logic. Because of this monolith, there is no privilege management, which means the impact of a vulnerability in any component is catastrophic; the entire device's code is the Trusted Computing Base (TCB).

[2]The IETF defines class 1 IoT devices as having ~10 KiB of RAM and ~100 KiB of storage, and class 2 devices as having ~50 KiB of RAM and ~250 KiB of storage [25].

On low-end IoT devices, firmware images are often purpose-built due to the one-off nature of IoT device hardware. While more powerful IoT devices will typically employ general purpose operating systems such as Linux [115], these devices are out of scope; Linux's heavy focus on preserving user space Application Binary Interface (ABI) compatibility [111] and backward compatibility means that many decades-old devices running Linux continue in operation.

Managing IoT devices requires keeping firmware up to date. Firmware updates are crucial for addressing security issues, enhancing device performance, and introducing new features. However, updating firmware on IoT devices can be difficult, particularly for devices with limited resources [19]. Challenges like storage capacity limitations, intermittent connectivity, and power constraints can hinder the firmware update process [56].

4.2.1. Software Update Schemes for IoT

Several state-of-the-art firmware update systems have been developed to tackle the challenges faced by resource-constrained IoT devices [81, 112, 101]. These frameworks offer solutions that address the constraints of such devices and mitigate various threats. However, one significant limitation of current state-of-the-art designs is their lack of consideration for the long-term deployment model of IoT devices. There has been limited exploration of how these schemes will operate several decades into the future.

Efforts have been made to broadly analyze the longevity of IoT devices and the technical challenges associated with long-term deployment models [124, 15, 63]. These analyses cover issues related to longevity, but there is a lack of implementations that specifically address these challenges. This gap exists because many of these challenges extend beyond the scope of software update standards. We believe that a comprehensive and holistic perspective on IoT device security is required, along with potential architectural and paradigm changes to effectively tackle these challenges.

Most existing update frameworks designed for IoT devices rely on symmetric encryption, message authentication codes, and digital signatures to protect firmware in transit and verify it on the target device. However, these update schemes fail to consider how time will impact the overall security of the cryptographic algorithms they rely upon [63]. Kiningham et al. [66] discuss this issue while analyzing the potential for creating IoT devices with a 20-year lifespan. Indeed, cryptographic algorithms have gone from state-of-the-art to insecure in less than 20 years.

The Internet Engineering Task Force (IETF), acknowledges this problem in their recent proposal for Software Updates for the Internet of Things (SUIT). They emphasize that developers "must carefully consider the service lifetime of their product and the time horizon for quantum accelerated key extraction" [81] while implementing their update scheme. The worst-case estimate for the time horizon for quantum-accelerated key extraction is approximately 2030 [62]. Assuming the worst-case estimate holds true, this means that IoT devices created today may only have 7 years before they need

to be updated to support stronger cryptographic algorithms. Otherwise, the underlying communication channels they rely upon may be compromised. Note that SUIT appears to only consider issues related to asymmetric key attacks enabled by quantum computing, but does not comment on the plethora of non-quantum issues (e.g., inadvertent key leakage and revocation, implementation bugs, under-specification of protocols, etc.) that impact cryptographic systems today.

4.2.2. Device Vendors as a Single (and Complex) Point of Failure

Software update schemes for IoT devices tend to be designed under the assumption of a single entity responsible for building and distributing device firmware. We refer to this vendor as the *first-party vendor*, which is the original creator of an IoT device, and the entity responsible for creating and distributing firmware updates for an IoT device. Note that in more complex multi-stakeholder scenarios, such as if an IoT device is created and developed by one vendor and re-branded under another vendor, the original device vendor takes precedence. While this simple centralized model facilitates the development and suits the typical monolithic firmware design that is commonly found on these devices, it does not consider any form of device autonomy outside the walled garden it was designed within, thus establishing a one-to-one relationship between a device and its first-party vendor.

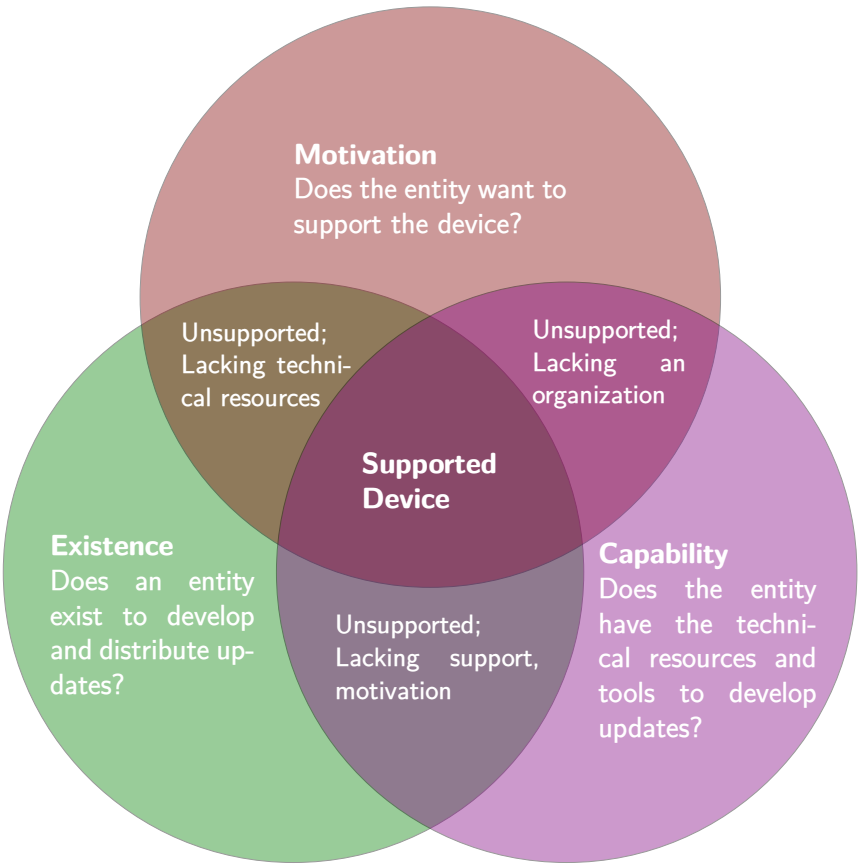


Figure 4.1.: Whether a vendor can provide software updates depends on their existence, motivation, and ability. Without any one of these factors, a vendor's ability to support devices becomes hindered.

Under this one-to-one device-to-vendor model, the issue becomes evident: if the vendor disappears, IoT devices managed by that vendor will no longer receive updates (see the bottom left circle in Figure 4.1). Even if the vendor continues regular operations, they must be motivated *and* retain the technical and human resources to develop updates for specific devices. Only when these three dependencies (existence, motivation, and capability) are met can a device be considered supported by the vendor.

The remaining intersections in Figure 4.1 reveal a range of factors that can prevent updates from being built. For example, if an organization is motivated but lacks the necessary developer resources, has accumulated excessive technical debt, or has lost access to tooling, they will be unable to distribute updates. Similarly, if a vendor may have the capability but lack motivation to issue updates; the vendor may not have sufficient financial incentives, or the business may have decided to prioritize support for other product lines. Finally, if a vendor has the motivation and resources to produce updates, but ceases to exist (e.g., the company goes out of business), no further updates will be produced.

Considering each of these factors as a point of failure, it becomes apparent that the first-party vendor producing any software, let alone updates for a legacy device is a colossal task. Each of these factors becomes more of a concern with IoT: for devices that are being deployed into permanent installation settings with long-term lifespans, this is a exceedingly fragile ecosystem.

Within this context, we propose a new way to think about IoT software updates, which to our knowledge has not yet surfaced in the IoT security literature: *the first-party vendor is a point of failure.* To allow long-term use and durability of IoT devices, their software needs to be updated. Devices should not rely exclusively on the first-party vendor for updates.

4.2.3. Software Updates within Walled Gardens

On Personal Computers (PCs), users do not rely on the manufacturer of a device's hardware to build, distribute, and maintain all the software they need to operate the device. Instead, the PC vendor provides the hardware and maintains core pieces of firmware and software as needed: the Basic Input/Output System (BIOS), system drivers, and controller drivers, among other things. The applications and software are distributed by other sources.

Although the model of obtaining software from a variety of sources is widely used in the world of general-purpose computing devices, it is worth noting that this model is not universally adopted. The converse model is most obviously apparent in smartphones, which use a more heavily centralized software distribution model. While applications for smartphones are developed by different independent sources, they are distributed through centralized sources that are controlled by the phone manufacturer or operating system vendor. An example of this is the Apple App Store, which is used by iPhone and iPad devices [11]. A contrast to this is the Android

model, which demonstrates that full centralization and lock-in is not the only approach. Android does not lock users to a single centralized app store. Users are free to use third-party app stores, such as F-Droid[3], thus enabling an approach that favors user control.

The "walled gardens" offer higher levels of control that can be beneficial for security reasons[4], but it also raises concerns about monopolistic practices and, more importantly, the single point of failure it creates. If the entity maintaining a centralized "walled garden" app store were to suddenly disappear, users would not have any options for receiving software updates, or for installing software at all. In an approach that favors user choice and autonomy, the disappearance of the entity maintaining a centralized app store would merely be an inconvenience, but not prevent users from using a third-party software distribution network.

The ongoing debate regarding these app stores relates to IoT: themes of centralized control versus individual autonomy underpin current events. It highlights the need for new paradigms that can accommodate both the benefits of centralized control and the benefits of individual autonomy.

[3] https://f-droid.org

[4] Centralized app stores can perform application vetting, including developer authentication, functionality checks, API tests, security tests, etc. [18]

4.3. Non-Solutions to IoT Longevity

In this section, we examine existing methods and measures that aim to extend the lifespan of IoT devices. It is often suggested that these methods can solve some problems discussed in Section 4.2; however, we believe that the ideas listed below only shift technical burdens onto other parties without addressing the underlying issue. It is worth noting that some of these methods may still be useful for improving vendor incentives (specifically 4.3.1 and 4.3.2), establishing standardized mechanisms for secure firmware distribution (4.3.4), or delegating legacy maintenance to third parties (4.3.3 and 4.3.5). The non-solutions mentioned below are not an exhaustive list of all the alternatives considered to date. Our focus is on notable suggestions that highlight the shortcomings of current models.

4.3.1. Software Updates as a Paid Service

A potential solution for those wishing to have devices updated beyond the manufacturer-supported cost-free period is for the original device manufacturer to offer software updates as a paid service [113]. This is a common practice in corporate and enterprise software (e.g., Red Hat Enterprise Linux). Users who pay for a license (that can be paid e.g., monthly or yearly) receive software updates for their devices. Microsoft famously continues to support the 22-year-old Windows XP for enterprise customers who are willing to pay for support [59].

The benefit of this approach is it provides an economically sustainable way to prolong device updates from the device's original vendor. Many IoT device vendors currently provide device updates at no cost to the user, but eventually, it stops making financial sense for vendors to keep throwing development resources at products that no longer generate income. Figure 4.1 highlights this in terms of vendor motivation; an additional income stream can motivate vendors to develop updates for long periods.

Unfortunately, while this general idea may address vendor motivation, it still requires the first-party vendor to be available and to be technically capable of performing device updates. If the first party ceases to exist, these "update subscriptions" will terminate and the legacy devices will progressively age and fall out of date, resulting in the original problem of vulnerable and unsupported devices.

While updates as a paid service can be an effective solution to extending the period a vendor is motivated to provide firmware updates, it is not a solution that allows for delegation of responsibility. The first-party vendor will remain the single point of failure. Manufacturers need to ensure long-term support for their devices, so there must be a contingency plan in place in case the manufacturer is unable, unmotivated, or nonexistent.

4.3.2. Device Leasing Model

In enterprise and corporate settings, equipment leasing is a common practice where hardware is rented or leased under a service agreement. At the end of the agreement,

the vendor responsible for the device will typically replace it with new hardware. This way, the service provide ensures that the customer always has access to the latest supported hardware covered under a service agreement. After the device is decommissioned, it may be refurbished and re-sold as a previous-generation device. The sale of used equipment is a great opportunity to reuse a device instead of throwing it away to be recycled; however, the re-sale is only possible if the device has some value at the end of its lease. This is particularly applicable to valuable enterprise-grade hardware such as servers, network switches, and workstations.

Not all IoT devices can be leased, especially consumer devices such as smart home gadgets and wearables. These devices are usually owned outright by the users. Leasing or renting IoT devices may also be too expensive for some use cases, especially in consumer and industrial markets. For devices that require professional installation or are encased in homes, replacing them can be a hassle for consumers and offer no significant improvement in longevity.

While this does provide the end-user with supported hardware which will receive updates, this model does not address the IoT longevity problem as it only pushes the burden of legacy device maintenance to the new device owner. Once a service agreement ends and devices are decommissioned, they may be recycled and refurbished to be sold on a third-party market.

4.3.3. Release of Source Code and Tooling

Another potential approach is to require (perhaps through legal means) the first-party vendor to release all the code and tooling for the device such that a third party to continue maintenance and development. This third party could be another company or an open-source community that is willing to take on the development responsibility. While this solution appears beneficial, it faces several practical challenges. For example, some manufacturers may not have the legal right to release the source code for their devices due to proprietary software or intellectual property issues. Additionally, third-party developers may not have the expertise or resources to effectively support older devices, which could lead to security and compatibility issues.

While we strongly advocate for open implementations, this method only shifts the maintenance burden to another entity. This could result in a few maintainers being responsible for a vast number of diverse IoT device firmware codes. Over time, compilers and build tools will become outdated, causing a decrease in the number of developers who can maintain these codebases. As maintainers shift their focus to newer devices, older ones will eventually become outdated.

4.3.4. Unified IoT Protocols

Several protocol designs for IoT devices aim to provide a standard set of abstractions to IoT device functionality. Protocols such as LWM2M [112] provide standard ways

to provision IoT devices, send/receive data, and more importantly perform software updates, which is a comprehensive holistic protocol for an IoT device's lifecycle. Other protocols aim to provide unified solutions for only firmware updates, such as IETF SUIT [81].

Unfortunately, these protocols deem the issue of device longevity out of scope [81], or ignore it altogether [112]. Longevity is a crucial issue for these unified protocols, which directly impacts the effectiveness of the cryptographic primitives used to provide security and integrity to firmware updates and ensure the device can operate securely [81, 19]. These protocols do not cover what should be done with legacy devices that do not support current cryptographic algorithms, leaving it up to the device manufacturer to make these decisions. Without adequate training, context, and supporting infrastructure, leaving these critical decisions to manufacturers (or more specifically, developers employed by the manufacturer) is unlikely to result in more secure software [121].

Additionally, proposals that cover firmware updates only consider firmware originating from a first-party vendor, which impacts long-term security and trust. These proposals do not have provisions for vendor agility, forcing first-party vendors to use unspecified, ad-hoc out-of-band processes to hand off support to third parties.

Unified IoT protocols are a non-solution for increasing device longevity – they do not consider factors that ultimately impact long-term device longevity [63, 15] as the various challenges impacting device longevity are out of the scope of what

these protocols aim to achieve. Thus, protocols are certainly part of the solution to long-term device updates. Creating consistent APIs for IoT devices to conform to will greatly assist in creating a unified and consistent way to distribute firmware to heterogeneous IoT devices.

4.3.5. Open-source IoT Frameworks

The open-source movement has influenced IoT hardware vendors, leading to efforts to make their device development frameworks and SDKs open-source. A notable example is Espressif, a vendor who has embraced open-source as part of their development model. As a result, open-source communities have emerged around their SDKs and IoT development boards, leading to the creation of projects like ESP Home[5] and Tasmota[6]. Both of these projects allow IoT enthusiasts to write custom firmware for Espressif devices.

Despite the benefits of this strategy, there are some disadvantages to slowly adopting an open-source model. For instance, developers must use Espressif's fork of Low Level Virtual Machine (LLVM)[7] to utilize any of their SDKs on Espressif boards that utilize the Xtensa architecture. If Espressif fails to keep its LLVM fork up to date (it is currently approximately 10,000 commits behind upstream LLVM), its compiler infrastructure for Xtensa boards may become stagnant. Nevertheless, Espressif has

[5]https://esphome.io
[6]https://tasmota.github.io
[7]https://github.com/espressif/llvm-project

actively worked on adding this LLVM backend to upstream LLVM for approximately four years, suggesting that they may eventually embrace open-source collaboration.

The concerns raised regarding tooling and dependencies highlight several challenges in the software supply chains of IoT devices. The firmware for such devices often relies on external dependencies, which in turn creates intricate supply chains with interdependencies between various vendors and organizations. Each participant in this chain introduces a potential point of failure or vulnerability that may impede the timely and secure delivery of software updates to IoT devices. While the adoption of a fully open-source model for IoT device development could help with some of these challenges, the presence of closed-source or proprietary dependencies may hinder the efforts of open-source communities.

While open-source communities cannot solve all the issues related to IoT device firmware, they still have an important role to play. We believe that these communities can serve as a valuable third party for intervention, helping to ensure that IoT devices remain secure and up-to-date in the face of rapidly evolving threats. At the same time, we also acknowledge that some proprietary dependencies may be necessary in certain cases. While we would ideally prefer an entirely open-source model for IoT device development, we recognize that this may not always be practical or feasible. We believe that architectural changes to IoT device firmware are needed to accommodate these dependencies and ensure that they do not become a barrier to effective device management. These changes are discussed in greater detail in Section 4.5.

4.3.6. IoT Recycling

A common pattern seen in other hardware industries such as general-purpose computers and mobile phones is taking functional, yet unsupported, hardware and recycling it responsibly, such that new devices can be made sustainable with resources extracted from old devices, thus creating a more sustainable circular economy [43]. With this in mind, a common suggestion is for IoT devices to adopt a similar e-waste recycling strategy [113].

The issue of IoT recycling and e-waste recycling more broadly is characterized by a significant global inefficiency in waste management. According to the 2019 Global e-waste monitor, a staggering 53.6 metric tonnes of e-waste was generated worldwide, with only 9.3 metric tonnes (equivalent to approximately 17%) being adequately recycled. The remaining 44.3 metric tonnes (equivalent to approximately 82%) were either dumped, traded, or recycled in a manner that is not environmentally sustainable [51].

There are many reasons for the low utilization of e-waste recycling systems, but the main driver is usually the high cost of recycling these materials [51]. This challenge places significant financial pressures on the recycling sector, which may limit its capacity to properly recycle e-waste. Moreover, IoT devices are not attractive targets for recycling. These devices are small, cheap to mass-produce, and difficult to recycle. Separating recyclable from non-recyclable materials in IoT devices is far from trivial, and due to their small size, any valuable resources that can be extracted is small [51].

Even when materials can be recovered and recycled, the resulting product may not always be suitable for the same purpose as the original device. For example, it's more cost-effective to use recycled plastics as insulation rather than the creation of new plastics for the same original purpose.

Societal and cultural aspects of recycling must also be considered here. Users spending a few dollars for an IoT device may not feel motivated to drive[8] to their nearest recycling facility to responsibly dispose of the device. Storing unsupported IoT devices in the home for future disposal can also be dangerous; coin-cell batteries can be deadly if ingested, and other batteries may leak and cause damage. As discussed in Section 4.7.3, for recycling to work, IoT vendors need to be held accountable not only for the production of long-lasting devices but also for their proper disposal.

4.4. Longevity and Durability in IoT Device Software

To make progress towards increasing IoT device longevity, we must first understand where in the IoT device lifecycle longevity may be unnecessarily limited. We begin by drawing a distinction between *longevity* and *durability* [43, 37], and then review how obsolescence plays a role in each of the stages of the IoT lifecycle.

Longevity refers to the period during which a device is useful from the time it is sold

[8]For low-cost devices, the cost in the fuel may be greater than the production cost of the device.

until it is disposed of or replaced. Note that "usefulness" is a largely subjective factor that is dependent on the end-user of the device [38]. These subjective factors include perceived product characteristics, situational influence, and consumer characteristics. For example, consumers may purchase a new smartphone every few years to benefit from the newest features despite owning a fully functional device – these replacements can occur while the product is well within its working lifetime [49]. Situational influence, such as the emergence of new technologies or the changing economic landscape, can also impact a device's longevity. Lastly, consumer demographics, such as age, income, and location, can influence how consumers perceive and use IoT devices, which can impact the device's lifespan.

Durability, on the other hand, is the intended period for which the device was designed to be functional, as specified by the device's designers and engineers. Factors driving durability tend to be more objective than those seen in longevity (e.g., type and quality of materials used, manufacturing process, expected wear and tear of the device, etc.), and such factors are driven by consumer requirements and expectations.

To summarize, longevity is largely impacted by consumer-oriented subjective factors, while durability is impacted by product-oriented objective factors. There is a significant overlap between subjective and objective factors: for example, product requirements are objective factors; however, product requirements are derived from the largely subjective needs of consumers. Furthermore, the objective factors that impact device durability directly impact device longevity: if a device is not durable,

the useful lifetime of a device is significantly reduced.

The literature uses the terms longevity and durability to describe products holistically [38, 49]; however, in the context of device software specifically, longevity, and durability also play a role. When discussing software, longevity refers to the software's usefulness, which depends on the features it offers and how those features are presented to the consumer. For instance, a printer compatible only with the AirPrint protocol would be useful solely to users with AirPrint-capable devices. Should users switch to devices lacking AirPrint support, the subjective value of the printer would diminish (despite the printer not changing whatsoever). Thus, we believe that to ensure the product remains useful for an extended period, the software for that product must be able to evolve to meet consumer needs.

The durability of software relies on its correctness. In essence, any software bugs undermine its correctness and, consequently, its durability. A subset of these bugs would be security-related, such as software vulnerabilities, thus making software security an objective factor that directly impacts software durability.

4.4.1. Obsolescence and the IoT Lifecycle

This section presents the IoT device lifecycle, as shown in Figure 4.2, with an emphasis on various degrees of obsolescence. We begin by discussing device inception (Stage 0), and initial deployment by a first-party vendor (Stage 1). Then, we consider the various stages that occur after support has ended: programmed obsolescence (Stage

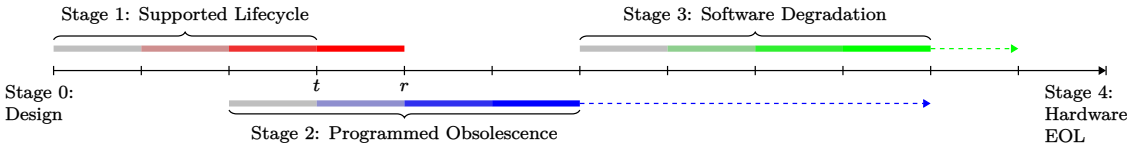


Figure 4.2.: An IoT device's lifetime represented as an abstract timeline. From device inception, the first period t represents the end of software support from the first-party vendor, and r represents the point in time when a lack of software updates causes a degradation in device functionality. We propose adding a transition period where an IoT device can detect obsolescence, and then transition to a third-party development model.

2) which will eventually lead to software degradation (Stage 3). Finally, we reach the end of life for an IoT device, when the hardware can no longer sustain device functionality (Stage 4).

Stage 0: Design

The creation of an IoT device begins with the first-party vendor. This vendor is responsible for designing the device, including its intended function and the hardware and software required to make it perform that function. At this stage, whether explicitly or implicitly, critical decisions are made about the device's longevity and durability. For instance, a vendor may decide an IoT device should have a lifetime of 5 years, and thus be engineered to have at least 5 years of durability under normal usage [100]. These decisions directly impact the overall durability of the device; however, just because a device may be designed with durability in mind does not

imply it will be useful for the intended service lifetime.

A lack of consideration for long-term software updates impacts device longevity. The vendor must therefore decide whether the device will receive software updates, and if so, select or build their own update infrastructure. The IoT industry to date has offered many examples of software updates being an afterthought, leading to insecure, incomplete, or ad-hoc choices for distributing updates. Empirically, the velocity with which new devices are shipped appears to take priority over the design of a sound software update architecture [29].

Finally, a critical decision that the vendor must make is what happens to the IoT device once it is no longer supported. Murakami et al. [84] refer to this decision as *planned* or *programmed* obsolescence. If the vendor has considered this scenario, they can inform consumers of the device's end-of-life date or provide a path for continued use. We augment the terminology of Murakami et al. to include *negligent* obsolescence: when a vendor avoids making this decision entirely, leading to the inevitable reduction of the device's lifespan.

Stage 1: Deployment, Supported Lifecycle

The IoT device is designed and ready for release, at this point the device will be deployed by an end-user and brought online. From now until the end of support any bug fixes, patches, and features, are delivered through the first-party vendor. This stage will continue on for the supported lifespan of the device, or until the vendor

loses the ability, motivation, or ceases to exist. The vendor may have a predetermined timespan for software support (e.g., 5 years), or the vendor may have a paid model for receiving updates. The service life from the previous stage should be how long the device is supported before various types of obsolescence set in.

Throughout this supported lifecycle, various forms of *relative obsolescence* may occur. Relative obsolescence refers to the disuse of a functional product, which may occur due to several factors [38]. Subjective factors are going to determine if a device will enter relative obsolescence. From a software perspective, this can be seen as the integrations and software features supported by the IoT device: if a consumer has an IoT device that does not support Android integrations, and the consumer migrates to an Android-only ecosystem, then the device will technically be functional but not useful to the consumer.

Stage 2: Unsupported, Programmed Obsolescence

Eventually, the firmware that runs the device will become unsupported. In Figure 4.2, this is shown as marker t, which represents the end of first-party support from the first-party vendor (e.g., through the factors in Section 4.2.2). This is where choices made in Stage 0 become relevant: what will happen next for the device? If no consideration of the device's software afterlife was made during the device's initial design, there is a chance the device won't make it past stage 3 (i.e., due to negligent obsolescence). The device may retain some of its original functionality; however, any

lingering vulnerabilities may be left unpatched for an undetermined amount of time.

A recent example of this is monitor-io, an IoT device for monitoring the quality of home Internet connections. When the monitor-io device maker closed its doors it provided its users with a standalone firmware image that will allow their devices to keep functioning in the absence of the vendor's hosted services [46]. While this option will extend the lifespan of the device, it is unclear who (if anyone) is now responsible for maintaining and supporting monitor-io devices.

Another recent example is Amazon deciding in April 2023 to remotely disable their line of Halo fitness trackers in July 2023, encouraging users to recycle their devices [3]. Here, the fitness trackers will not experience any software degradation as explained below since they will no longer be functional in any way.

Note that the term "programmed obsolescence" can be used to describe multiple ways that software can lead to device obsolescence. Programmed obsolescence is most commonly used to refer to intentionally writing software that limits device functionality [116, 42]. Instances of this can be created by vendors by pushing a software update that disables features, slows down device performance, or by making the device unusable [83].

Another form of programmed obsolescence arises from external factors. If an IoT device relies on external services for any functionality and the service introduces a disruptive change to its API or stops working, this represents a distinct type of programmed obsolescence. To maintain clarity we refer to this type of programmed

obsolescence as *software degradation*. In these instances, it is not the intentional inclusion of life-limiting features by the original device vendor, but rather an external dependency that knowingly or unknowingly impairs functionality. These environmental changes, despite being external to a device and the device's software, impact software durability.

Stage 3: Software Degradation

With the onset of time, the device will continue to work with the latest firmware build that was installed. Depending on the evolution of the protocols and standards that the IoT device depends on, the device may retain some level of functionality so long as the various layers of the environment remain compatible. We show this in Figure 4.2 as the period between t and r, and note that this period may not be linear. An unsupported IoT device may continue to work for several years after becoming unsupported. Several forms of software degradation can occur during this stage, as we explain in Section 4.5.2. For example, the IoT device may not support the required cipher suites to connect to newer TLS endpoints [29], causing failures in remote data retrieval. Another example is the Y2K38 bug (expected on January 19, 2038) that will affect devices that still use a 32-bit integer to track the number of elapsed seconds since the epoch [52]. This will cause time-based errors and inconsistencies on devices that have not migrated to a 64-bit integer for storing the time. The devices that do not retain their original purpose (despite having functional hardware) will likely be

thrown out, contributing to an ever-growing e-waste problem.

Stage 4: Hardware End of Life

Eventually, physical hardware will degrade to a point where the device can no longer physically function. Flash memory will wear out after too many write cycles, and electronic components will degrade beyond their tolerances. At this point, the hardware can no longer serve the end user, and at this point, the device will turn into e-waste. Ideally, if the device was originally designed with durability in mind, the hardware deterioration will occur after the device's expected service lifespan. Once the hardware has reached this stage, it can be interpreted as *absolute obsolescence* [38].

Once device hardware no longer functions, the next path it takes is dependent on the economic model it is created in. As we discuss in Section 4.7.2, in a traditional linear economy once hardware reaches its end of life it will be disposed of. This is the most common path IoT devices take, a total loss scenario where none of the resources inside the device are re-purposed[9], and new devices are made from our planet's finite resources. This is contrasted by a circular economy, where efforts will be made to repair broken devices and recover precious materials to produce new devices. By embracing circular economy principles for IoT devices, we can not only reduce waste and environmental impact but also create a more sustainable and efficient system.

[9]Due to the challenges discussed in Section 4.3.6.

The end-of-life of one device can become the starting point for another, creating a closed loop of resource use that benefits both consumers and the planet.

4.5. Towards IoT Device Longevity

In this section, we compare historical advances in computing with IoT devices to better understand the unique challenges faced by the IoT industry. We examine the factors that allow for long-lasting systems in a general sense, taking examples from computing devices that date back nearly half a century. We use these factors to identify the main limiting factor of IoT device lifespans: the internet, along with the inherent complexity and points of failure that internet-connected devices have to contend with.

4.5.1. What Makes a Long-lasting System?

The problem of longevity may stand directly in the face of IoT; however, there are several past examples of computer hardware that have withstood the test of time. Some of the first home computers from 40+ years ago such as the VIC-20, Apple II, and Amiga (among many others) still function today. Many collectors and enthusiasts keep these older machines running. We believe it is valuable to unpack some of the objective aspects that have contributed to their long lifespan.

Starting at the lowest level, these older devices have simple and modular hard-

ware made from commodity parts. Replacing a faulty chip with a working one is straightforward, as the internal hardware of these older systems relies heavily on socket-mounted integrated circuits. While hardware is not the primary concern of our IoT solution, we concur that hardware repairability is crucial for the durability of any device [43].

At higher levels, another key difference is that these devices were not heavily reliant on the internet for bootstrapping the operating system, or for installing applications. The norm for these machines was physical copies of the software, not ephemeral software distribution via app store. These machines were largely standalone in functionality with any internet functionality being a secondary feature.

4.5.2. The "I" in IoT Stands for Impermanence

Device complexity is a limiting factor of device longevity. Simple and minimal designs for both software and hardware are favorable when it comes to building reliable and long-lasting devices. One of the major limiting factors to IoT device longevity is the fundamental feature that sets IoT apart: the internet.

Software has become progressively more network-dependent with the advent of external vendor APIs and services, and this external dependency poses a problem; as external services roll out new changes and features, breaking changes become a common occurrence. Even following all the best practices regarding handling backward compatibility, at some point, legacy APIs will be deprecated and removed from

production deployments. This eventual deprecation of APIs has become part of the standard software development lifecycle and tends to not be disruptive as long as the consumers of these APIs keep their client code up-to-date.

The "I" in IoT is therefore a problem: we can create physically-durable things, but dependence on the internet inherently reduces software durability. Consider the following hypothetical IoT sprinkler: it supports programmable irrigation schedules and can retrieve precipitation forecasts from a public weather API. The sprinkler communicates with a client application on a smartphone through the vendor's cloud infrastructure. Over time:

- The vendor can no longer afford to run their cloud infrastructure, so they shut it down. App access to the sprinkler ceases to function, but users can still program the sprinkler by using the touchscreen interface on the device.
- The weather API updates their endpoints to be TLS-only. The sprinkler's limited TLS support does not have the root Certificate Authority (CA) certificate for the weather API, so connections to the API cannot be authenticated (but data can still be retrieved insecurely).
- The weather API updates to version 2 and deprecates version 1's endpoint. The sprinkler, unaware of the change, is no longer able to determine if it will rain in the near future.

- The hard-coded network time protocol (NTP) service used by the sprinkler to determine the current time goes offline. The sprinkler's internal clock slowly drifts, causing irrigation to begin at the wrong time. The user can manually reset the time but needs to do so every month.

Note that the failure modes listed above were all caused by external, internet-based dependencies changing or going away. The example highlights how even well-intended changes (i.e., supporting a secure TLS transport) are difficult to anticipate and support perpetually[10] without software updates.

To cope with the ever-changing nature of the internet and all the building blocks needed securely interact with it, firmware updates are a requirement for these devices. A firmware update infrastructure inherently adds new points of failure on all the external internet-connected dependencies, and if any building block in the stack has a breaking change, the entire stack can fail, hinting at the need for a robust, fault-tolerant update infrastructure.

4.5.3. Inspiration from Previous Paradigm Shifts

Considering the unique challenges of IoT, no existing research tool or industrial effort offers a direct solution to the longevity problem. However, the longevity challenges faced by IoT are not unique. A similar set of challenges plagued early personal com-

[10] Consider that not only is the root CA certificate required, but it also needs to be replaced when it expires.

puters[11] in the 1970s and 1980s. During this period, computers did not have general-purpose operating systems, instead using hardware-specific operating systems created by the manufacturer. For instance, computers made by Commodore could not run Apple's operating system, and IBM computers could not run Commodore's. The heterogeneity was mainly due to a lack of standardization between computers, operating systems, and hardware. In the following decades, general-purpose computers emerged, and operating systems became usable as long as they had support from the underlying hardware architecture. This transition led to a clear separation of responsibilities between the hardware creator and the operating system developer, which is something that IoT should aspire to.

We believe a general-purpose operating system for IoT is unlikely to see the same degree of adoption as e.g., Microsoft Windows or Linux due to the massive hardware heterogeneity in IoT[12]. Existing IoT operating systems such as Contiki [44], RIOT [14], Tock [70], among many others tend to support a narrow range of microcontrollers and hardware, with no cross-compatibility between OSes [54]. Applications built for any one of these IoT operating systems need to be largely rewritten to run on another OS.

Another paradigm shift occurred more recently with Android, the open-source smartphone operating system. Android was designed to allow smartphone manufac-

[11]The early days of personal computers that included operating systems.
[12]Manufacturers of microprocessors in the IoT space

turers to include their proprietary components and hardware support while offering a virtual machine runtime environment for user-space applications. Developers can target one of Android's Software Development Kit (SDK) versions and have confidence that their app will run on any Android phone (with heterogeneous hardware) that supports that version. This separation enables the creation of Android applications in a write-once-run-everywhere model.

The paradigm shift with Android is a partially attractive solution for IoT. Namely, the architectural movement that abstracts away hardware-specific details enables developers to write applications that target a standardized runtime. Android partially mitigates our concern with the first-party vendor being a point of failure: the OS for specific devices is built and distributed by the first-party vendor[13], but applications for Android can come from Google's play store or any third-party app store the user wishes to use. If the first-party vendor of an Android device stops updating the OS, this does not prevent applications from being updated so long as the applications remain compatible with the SDK compatibility level of the Android runtime on the device.

The concept of longevity in the context of Android extends beyond the domain of first-party vendors. Community-driven projects, such as LineageOS [1], have emerged within the Android community with the objective of extending software support to Android devices that are no longer officially maintained. This is achieved through

[13]The first-party vendor of the Android device itself, e.g., Samsung.

the collaborative efforts of open-source developers who create customized versions of Android, allowing users to install them on unsupported devices via an over-the-wire (One-Time Write (OTW)) update, thereby revitalizing their functionality. The ability for users to unlock their Android devices and replace the operating system is a key factor facilitating this project's success. For example, releases of Android 11 have been ported to many phones from 2014 by the LineageOS community [1].

One major obstacle in applying this approach to IoT devices is the lack of standardization for flashing firmware; no universal approach currently exists for users to connect and install the firmware using traditional OTW methods. Indeed, some IoT vendors conceal, make inaccessible, or disable hardware serial interfaces to prevent unofficial firmware flashing. While Android phones require some type of Universal Serial Bus (USB) port, IoT devices do not have such a requirement. For these IoT devices, the only firmware re-flashing option is an over-the-air (Over-the-Air (OTA)) updates, which, as previously mentioned in Section 4.2.1, does not allow for end-users to control device software, and is limited to the first-party vendor.

Additionally, even if a standard technical mechanism for flashing IoT firmware in an over-the-wire fashion emerged, there are practical challenges with having users access to the IoT device to perform an OTW update. This is especially important when IoT devices are installed inside appliances or walls, making it difficult for users to access the device to perform the update. Therefore, it becomes imperative to explore alternative solutions that do not rely exclusively on user involvement for re-flashing

the device firmware. Giving users such an option is beneficial, but it should not be the only solution.

4.6. A New Paradigm for IoT Device Longevity

The only way the Internet of Things can continue to evolve and be maintainable for long periods of time is to eliminate the long-term maintenance burden on a single entity, allowing for the responsibility of maintenance to be securely delegated to new entities. Specifically, our proposal involves clearly denoting the components and software that the first-party vendor is responsible, for and expecting that vendor to eventually disappear. If architected correctly, we believe an IoT device built with these principles can remain operational for the designed lifetime of the hardware.

4.6.1. Addressing the Maintenance Burden

In our model, the first-party vendor is initially responsible for all layers of the device, as shown in Figure 4.3. The first-party vendor needs to develop the hardware along with some primitives that would be implemented for most firmware designs: a hardware abstraction layer (Hardware Abstraction Layer (HAL)), any proprietary firmware needed to enable hardware functionality, and some basic primitives for a microkernel-inspired OS [14, 67]. As shown in Figure 4.3, this would correspond to the middle portions of the operating system stack shown in blue. These layers can ef-

fectively contain all of the first-party vendor's Internet Protocol (IP), any proprietary components and/or components that require proprietary tooling will be contained within the microkernel.

Building application logic using natively-compiled code negatively impacts modularity as discussed in Section 4.3.5, therefore, we propose a shift towards a generic execution environment that is agnostic of the underlying device architecture. IoT devices that require special compilers and tooling will ultimately mean that future maintainers have an additional burden to build and distribute binaries for these devices. Therefore, we envision a platform-independent runtime for all application logic, libraries, and abstractions beyond the microkernel. Additionally, we propose that the platform-independent runtime expose a standard set of APIs for interacting with the underlying device hardware. This is somewhat similar to the Android model discussed in Section 4.5.3, with one major change being a microkernel design. Android uses the monolithic Linux kernel, which not only increases the trusted computing base (TCB) but also increases the likelihood of eventual bugs. By reducing the size of the software provided by the vendor, we are effectively minimizing the need for any first-party updates beyond the first-party support period.

Our envisioned platform-independent runtime would expose a set of standard APIs for peripheral access such as Universal Asynchronous Receiver / Transmitter (UART), Serial Peripheral Interface (SPI), and flash storage. For the networking stack, we propose any proprietary firmware to be included in the microkernel, and standard

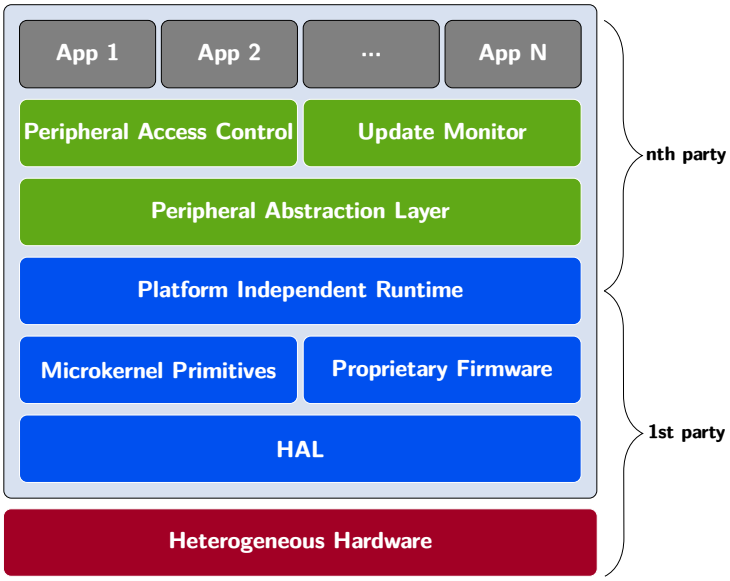


Figure 4.3.: Our proposed development model splits responsibility between multiple parties. The creator of the device is responsible for creating the device's hardware and implementing a minimal OS and runtime. Platform-independent code can be created for multiple devices without proprietary tooling.

abstractions for interacting with network devices be exposed to applications via the platform-independent runtime.

Microkernel: In our model, the first-party vendor is accountable for the microkernel and runtime components of the operating system. We deliberately select a microkernel design to keep the trusted computing base of the OS small and minimize complexity. A common drawback of microkernel designs is additional overhead due to a large amount of message passing and context switching compared to monolithic kernels [67];

however, IoT deployment verticals are typically not high-performance[14], therefore this appears to be a reasonable trade-off.

Even though the microkernel design can reduce the number of bugs and their impact on the rest of the device, it is important to note that bugs can (and will) still occur. To address this, we propose that during the typical deployment timeline of IoT devices, when the first-party vendor actively maintains the device (e.g., Stage 1, Sec. 4.4.1), any major bugs in the microkernel or runtime should be resolved within the period when the first-party vendor solely maintains the device. Ideally, only a small number of lingering bugs (if any) would exist in the device after the first-party support period ends, and these bugs will not be high severity[15]. If a high-severity bug is found in the kernel after the vendor support period has ended – which we believe to be less likely due to the microkernel's reduced size, responsibility, and complexity – the first-party vendor will need to fix and update the kernel. If the first-party vendor is no longer available, the device will need to be reverse engineered so that a new microkernel/HAL can be written from scratch for it, and flashed over the wire directly to the device.

Platform Independent Runtime: The platform-independent runtime enables applications, libraries, and device abstractions to execute in a user-space environment,

[14]There are exceptions of high-performance IoT devices such as control systems in planes and computer-vision enabled cameras; however, these are generally the exception to the rule, see Section 2.1 for more information.

[15]If the device is maintained relatively regularly by the first-party vendor, it should remain relatively secure barring any zero-day attacks that would impact that particular device.

without relying on any platform-specific code. This design choice solves the issues highlighted in Section 4.3.5: future developers will no longer be restricted by the custom compiler and tooling infrastructure or proprietary compilers that require licensing. While developers must still understand the underlying hardware, the runtime abstraction allows critical libraries to follow a write-once-run-anywhere model. This is especially important for third-party maintainers.

Regarding the choice of runtime, we have not yet settled on the ideal candidate; however, there is one particular runtime that stands out as an attractive option; WebAssembly is a portable binary-code format originally designed for execution within web browsers using a virtual machine. Over time, WebAssembly has expanded beyond the browser, especially with the introduction of the WebAssembly System Interface (WASI) [8], which enables access to lower-level OS and hardware features. The emergence of the lightweight WebAssembly Micro Runtime (WAMR) [33], optimized for embedded devices further supports the viability of WebAssembly for our purposes.

By leveraging WebAssembly (and by extension WASI and WAMR), we can explore the potential advantages this runtime in our overall architecture. However, it is essential to conduct further research to determine the feasibility and compatibility of this approach in practice. For example, while there have been some feasibility studies on the use of WebAssembly on embedded devices [76, 120, 77], larger scale studies covering more heterogeneous hardware are needed to identify any shortcomings with the platform-independent runtime, the standardized API for interacting with device

peripherals, and with the separation and modularity of our OS design. Ensuring that this design is feasible on different classes of IoT devices [25] is crucial for its success.

One limitation of the platform-independent runtime as we envision it is complexity. While our approach limits the amount of complexity in the TCB through the use of a microkernel, some of that complexity cannot be avoided, and must be moved elsewhere on the stack; in this case, that complexity is moved to the platform-independent runtime. If any severe vulnerabilities are found in the runtime, in most cases only a first-party vendor will be able to provide a patch. However, in cases where manufacturers have an open-source development system and tooling as described in Section 4.3.3 and 4.3.5, an open-source community could provide updates to the platform-independent runtime. To support this, vendors should be encouraged to adopt proven open-source runtime implementations. In the case of WebAssembly, WAMR appears to be a good candidate as it is one of the main open-source reference implementations of an embedded WebAssembly runtime.

4.6.2. Detecting First-Party Vendor Failure

We now discuss strategies for enabling IoT devices to autonomously decide if their first-party vendor no longer provides support. This is particularly useful as it does not require the device owner to actively check if their (potentially dozens of) IoT devices are actively supported. IoT devices can use this knowledge to determine when they can transition to another support channel as described in Section 4.6.3.

Heuristic Name	**Type**	**Example**
Relative time	Static	5 years from deployment
Fixed time	Static	On January 1, 2030
Heartbeat	Dynamic	Check first-party API
DNS	Dynamic	Check first-party DNS record
Open collective	Dynamic	Check central device authority
Relative SLA	Hybrid	If 3 years since last update

Table 4.1.: Proposed vendor-liveliness heuristics that can be autonomously evaluated by an IoT device to test if a first-party vendor is still supporting a device. A static heuristic has no external dependency, a dynamic heuristic depends on a dynamic check on an external dependency, and a hybrid heuristic uses a combination of static and dynamic checks.

This connects to Stage 2 and Stage 3 of our model: instead of an unsupported device running progressively more outdated software, it can decide when to transition and switch to actively maintained software to avoid suffering from software degradation.

In our model, the vendor in charge sets liveliness metrics for the IoT device to detect if it is currently being supported by the vendor without any explicit notification or manual checking required from the user. These metrics ensure that the device can make independent decisions about ongoing support.

The specific vendor-liveliness heuristics will depend on the use case of the IoT device. A list of potential heuristics and examples is given in Table 4.1. For example, if a vendor expects to support the device for 2 years, they can use a static heuristic such as a timestamp stating the last day of support, allowing a device with a reasonably accurate clock to determine if it is within the support period.

For checks that depend on external resources, dynamic heuristics can be used. Dynamic heuristics rely on the device's inherent connectivity. For example, the device can check for a DNS record belonging to the first-party vendor, if the vendor no longer exists, these DNS records may no longer be available.

When considered individually, each heuristic may not provide accurate results, thereby resulting in false positives (i.e., the conclusion that a support channel is no longer available). To address this limitation, we propose the combination of multiple heuristics, employing both static and dynamic checks, to enhance their reliability. By utilizing redundant sets of heuristics that draw from different sources, we can mitigate potential false positives that arise with any given heuristic. For instance, while a static temporal-based heuristic can be undermined by an error in or by spoofing the device's internal clock, the addition of a dynamic heuristic, such as periodic checks to a first-party service, can counteract this risk by ensuring both heuristics evaluate simultaneously. The layering of multiple heuristics enhances the overall reliability of the heuristic system.

Finally, we also suggest a fail-safe manual intervention mechanism for end users. Ultimately, users should have the ability to control the firmware that runs on their devices, allowing a secure mechanism for users to control what firmware repositories a device can access is another type of transition method that we suggest.

4.6.3. Vendor Agility

Once a device has identified through some heuristic or measure that it is no longer supported by its current maintainer, a secure transition to a different support channel must take place. Fortunately, there already exist technical mechanisms to accomplish this, so there is no need to reinvent the wheel. The transition can be done through the use of software update repositories, as is common on modern operating systems. The OS maintains a list of the current update sources, which serve metadata including timestamps and version numbers, and when new versions of the software are available, they are downloaded and verified through the provided download sources. Later, users can switch to a different repository (e.g., one serving beta versions of software, or the same software hosted elsewhere) by modifying the repository URIs.

Repository-based update schemes with multiple stakeholders have been previously proposed for the automotive sector (c.f., Uptane [64]). Uptane relies on a central metadata repository that is responsible for tracking and verifying the authenticity of firmware updates. Metadata for firmware updates is signed by the first-party vendor, or a trusted third-party vendor. While Uptane was not originally designed for our proposed model where we assume the first-party vendor is a point of failure, it provides a promising starting point for implementation.

We are effectively envisioning a model where IoT devices no longer *permanently* belong to the walled gardens of their manufacturers. Instead, software update sources, including security core functionality are distributed from one of many possible sources,

enabling a more robust protection against single points of failure. Once the first-party vendor is no longer available, the device can switch (ideally seamlessly) to another source and extend its software support period. One advantage here is that it should be possible for new software update sources to be added through software updates, ensuring a long-lasting product support period.

4.6.4. Transition Security

In our design, addressing the challenge of long-term deployment models necessitates consideration of both transition security and the longevity of cryptographic algorithms. As mentioned in Section 4.2.1, the use of cryptography plays a key role in ensuring the integrity and confidentiality of software updates. However, many state-of-the-art cryptographic algorithms have not demonstrated longevity beyond a 20-year lifespan [66]. For this new paradigm to be effective for long-term IoT deployments, addressing this issue is critical.

One of the primary solutions to overcoming outdated and/or vulnerable cryptographic implementations is through cryptographic agility. In our proposed design, cryptographic agility would be supported through updates to libraries within the platform-independent runtime, separated from the microkernel. This approach ensures that these libraries are not limited to a specific vendor's implementation. By decoupling the cryptographic libraries from the firmware and encapsulating them as modular packages, we can enable independent updates to be made to these libraries

– the entire device firmware does not need to be rebuilt solely to incorporate a fix for a single package. This modular approach empowers IoT devices to automatically and when needed update the cryptographic libraries, eliminating the dependency on vendors to release patched versions and preventing devices from becoming stagnant while awaiting such updates.

Stagnant device firmware – Another security concern is that several issues can prevent IoT devices from performing updates, thus causing devices to miss critical firmware updates that are needed to retain compatibility. Perhaps they are deployed within a network that blocks all outbound communications, or they were shut down and set aside for a decade. These devices, with the onset of time, will likely end up with some degree of software degradation due to the lack of rolling updates. If and when these devices re-connect to the internet, there may be enough breaking changes and software degradation that prevents these devices from performing updates, as the levels of software degradation would prevent the underlying protocols responsible for ensuring confidentiality and integrity would no longer function.

In such scenarios, the inclusion of a fail-safe manual intervention mechanism becomes essential to ensure their protection. End users will likely always need a secure method to directly connect to an IoT device, granting them control over the enabled software repositories within the device. Moreover, they should have the ability to manually update the device to the latest available firmware, thereby enabling it

to operate on newer software versions that remain compatible with the surrounding environment. While the design and implementation of this fail-safe mechanism lie beyond the scope of this chapter, it is worth considering firmware update schemes for IoT that leverage the partial offloading of resource-intensive tasks to a trusted local device, such as a smartphone. Notably, UpKit [68] presents a promising candidate that aligns with this model and merits further exploration.

Trusting third-parties – Additionally, there are other security considerations to take into account involving *where* transitioned software originates from. Pre-transition IoT devices (Stage 1) have a centralized source of firmware that is trusted as it originates from the first-party vendor. Once a device transitions to third-party support repositories (Stage 2) the root of trust becomes more difficult to establish. It is unclear whether there should now be two roots of trust, a single one for the new vendor, or a new root constructed from some cryptographic signature over both entity's signing keys. In any case, the device must always be able to detect whether an update originates from an untrusted party. Determining whether an update from a trusted party is malicious (e.g., due to insider threat compromise of signing keys) is out of scope.

One potential solution is a centralized third-party firmware distribution service specifically created for IoT devices. This service will be trusted by IoT devices and will need a trust anchor to be maintained on the devices from inception. However,

this design has some drawbacks, such as the need for static trust anchors and the risk of key compromise [101]. Ultimately, a third-party centralized authority will allow for the better overall security of IoT devices[16]. Nevertheless, this approach also introduces new points of failure.

An alternative solution to the challenges posed by the centralized third party is full decentralization. IoT devices can be configured with an arbitrary number of repositories that are not managed or governed by a single organization, thereby mitigating the risk of a single point of failure. Additionally, a decentralized approach could provide more flexibility and autonomy to IoT device manufacturers and users, as they would have greater control over the firmware update process and could choose to fetch updates from a variety of sources.

User choice – To ensure a positive user experience, we propose granting users the ability to make choices regarding their device's software sources. In a post-transition state, the device would be initially configured to trust a predefined set of central known third-party sources. This configuration allows the device to maintain the security and integrity of its software. However, we recognize the importance of user freedom and acknowledge that some users may prefer to change and configure the software sources according to their preferences.

[16] Assuming the authority has the policies, resources, and procedures to prevent malicious actors from distributing firmware

4.7. Discussion

We now switch our focus to discussing how the effort to increase IoT device longevity through software updates fits within the broader conversations related to the right to repair and environmental activism.

4.7.1. Right to Repair

The Right to Repair movement aims to create legislation that enables consumers to repair and modify their products by removing barriers imposed by manufacturers to prevent unauthorized repairs [87, 100]. These barriers force consumers to seek repairs from the first-party manufacturer or a subsidiary (authorized by the manufacturer). Examples include manufacturers restricting access to tools and methods required to perform repairs, adding software locks (e.g., through encryption, trusted platform modules, remote attestation) that prevent unauthorized repairs, or hindering device functionality if an unauthorized repair has occurred.

While the broad scope of the right-to-repair movement captures many practices across many industries, we are focused on the unique challenges IoT devices pose for its success: IoT devices use one-off firmware, running on one-off hardware, and as a result, there is typically little support outside of the manufacturer's walled garden. In turn, this results in devices that are nearly impossible to modify/repair by consumers, instead requiring a community of highly-specialized enthusiasts.

The reasoning for this turns out to be simple: the IoT devices are created in response to consumer demand, and consumers are not demanding. Instead, the demands are low-cost, small, and easy-to-use devices that connect to the internet and perform some convenient tasks. To create small and convenient IoT devices, hardware is designed specifically for the use case of a particular device. Chips, flash, and other peripherals are soldered directly to a printed circuit board. Adding modularity (e.g., replaceable chips installed in sockets) increases overall device size.

In addition, the devices' enclosure can be difficult to open – impeding access to the device internals – as there was no intention of allowing repairability. Compromises need to be made during the design process to fit a product to consumer demands, and in IoT, these compromises are repairability and vendor dependence. We believe addressing these concerns is possible through hardware that is engineered to be repairable which we discuss further in Section 4.7.3, but it would likely drive up prices.

Manufacturers of products with embedded firmware rely on copyright law to prevent their code from being reverse-engineered and copied [87]. They tend to oppose the right-to-repair legislation arguing that if such consumer protections were in place, they would be unable to protect the intellectual property stored inside the devices. Despite this, a report on the right to repair from the Federal Trade Commission (Federal Trade Commission (FTC)) noted that current copyright law already allows the owner of a device to copy a computer program for maintenance or repair. Additionally, consumers are permitted to circumvent technological protection measures

to diagnose, maintain, or repair certain products [50, 4, 5]. Certain very specific situations permit the circumvention of technological protection measures (TPMs) to restore functionality, as demonstrated in legally acquired software for medical devices, video games, network devices, and various other specific contexts. These instances allow consumers, end users, and other authorized individuals to bypass TPMs, as outlined in the relevant sections of the United States Code of Federal Regulations [4, 5].

From a legal perspective, this issue presents considerable ambiguity. In the past, when devices did not incorporate embedded firmware that necessitated users to agree to a license for the device to operate (commonly known as a "shrink wrap license" [87]), the concept of device ownership was relatively straightforward. Either a consumer-owned a device, or they did not. However, with the advent of IoT devices, the question of what precisely a consumer owns becomes uncertain. Although a consumer may purchase an IoT device and physically possess its hardware, the extent of ownership in relation to the device remains unclear.

While it may be argued that individuals have purchased the physical hardware of the device, the extent to which they are able to control and use the device is limited. This limitation arises due to the fact that, unlike general-purpose computers, IoT devices are not designed to enable users to run any software they choose. Rather, IoT devices are typically equipped with disposable software that is required for the device to function. Unlike general-purpose computers, IoT devices are designed to run

specific software and are reliant on first-party vendors for functionality and security.

One common counterargument to this view is that individuals could hypothetically flash their own software onto the device, thereby taking full control of it. As previously discussed, achieving this level of control is not a simple task. This process typically involves specialized equipment and technical expertise, as well as a significant investment of time and effort. The bar to entry to make this hypothetical scenario possible is far too high.

The limitations on control over IoT devices are intentional, not accidental. Manufacturers aim to create walled garden ecosystems to maintain dominance and profitability. In terms of ownership, possessing the physical device grants physical access but does not provide control over the firmware, unlike general-purpose computers.

4.7.2. Towards a Circular IoT Economy

A circular economy is an economic model that seeks to maximize the use of resources and minimize waste by keeping materials in use for as long as possible. In a circular economy, resources are used in a closed loop, where waste is minimized through recycling, reusing, and remanufacturing, rather than disposing of them after a single use [31]. The Internet of Things is far from a circular economy, it is instead a linear economy. In a linear economy, raw materials are extracted from the environment, processed into products, and eventually disposed of as waste after their use is no longer needed [31]. This approach assumes that resources are unlimited and the

resulting waste from the process can be easily absorbed by the environment. The linear economy operates on a throwaway culture where products are designed to be used once (and for a short period) and then discarded, resulting in a constant need for new resources and a growing amount of waste. This system leads to the depletion of natural resources, pollution, and other negative environmental impacts.

The manufacturing industry has been criticized for promoting the illusion of environmentally conscious decisions rather than implementing solutions that would actually help the environment. Companies like Coca-Cola, for instance, have been accused of greenwashing, which is when an organization spends more time and money on marketing itself as environmentally friendly than on actually minimizing its environmental impact [30, 93]. In fact, as of 2021, the Coca-Cola company ranked as the top global polluter of plastics for four consecutive years, emphasizing the significant contribution of its products to the plastic pollution crisis [30]. Instead of taking responsibility and implementing changes (e.g., moving away from plastic bottles) Coca-Cola's "green" marketing campaigns try to push the responsibility toward consumers and municipalities, arguing that they can't be held responsible for what people do with their product after purchase.

In the face of the challenge of ensuring the longevity of IoT devices, vendors may resort to greenwashing, and, similar to Coca-Cola, attempt to blame consumers and municipalities for the lack of high device recovery and recycling success. To avoid this, a paradigm shift is needed. While some vendors may initially struggle with the

proposed changes, a legislative intervention that takes into account all stakeholders is likely necessary to achieve meaningful change. This shift must address the entire lifecycle of IoT devices, including their manufacture, usage, and disposal. In this way, vendors can be held accountable for their environmental impact, and consumers will have access to accurate information that allows them to make informed purchasing decisions. Comprehensive legislation can level the playing field for all stakeholders, promoting sustainable practices and ensuring the longevity of the IoT industry.

4.7.3. Sustainable Design

Designing for sustainability can help IoT devices last longer, even after they become obsolete. Stead et al. [106] propose a sustainable design philosophy for IoT devices that aims to create devices that last a lifetime by being modular and repairable. According to this philosophy, if any part of the device breaks, it should be easily repairable by the end user, with minimal waste generated from the repair process. This philosophy applies to the embedded hardware board inside the device, meaning that the device (e.g., a toaster) should still function even if the microcontroller responsible for IoT functionality fails. To achieve this, the design should be modular, and a standard interface between the microcontroller board and the underlying device hardware should be established, eliminating the need for one-off implementations of IoT boards for heterogeneous products. This approach ensures that end-users can easily replace the microcontroller board, thus extending the life of the device.

Sustainable designs align with our proposed model, as it promotes the development of modular and sustainable firmware for embedded controllers over a long period. This would allow for the creation of standardized boards that can enable IoT functionality in various devices. For instance, the smart toaster [106] developed by Stead et al. can be sold without the embedded board responsible for enabling its IoT features. Consumers could then upgrade their toasters easily by purchasing the board separately if they desire the functionality. The boards could be designed to be generic, with the ability to recognize the type of device they are controlling and subsequently identify the appropriate firmware packages required for the device.

By designing IoT devices to make them sustainable, manufacturers can reduce electronic waste, reduce the need for frequent upgrades, and ultimately provide greater value to their customers. However, for this to become a widespread practice, manufacturers must adopt a comprehensive approach to sustainable design, with consideration given to all stages of the product lifecycle, from design to end-of-life disposal. Such an approach can only be achieved through a concerted effort by all stakeholders, including manufacturers, policymakers, and consumers.

4.8. Conclusion

Keeping IoT devices secure and functional over multiple decades is undoubtedly difficult. The reliance on Internet-based dependencies makes IoT device software degrade

over time. Without proper long-term vendor support, these devices are likely to end up in landfills even if the hardware remains functional. This chapter has argued that security plays an important role in the IoT device lifecycle and that the evolution of security protocols and algorithms necessitates a robust and decentralized software update infrastructure for IoT. A initial software stack for future devices was proposed along with a set of technical mechanisms through which devices can securely switch to a new support channel, once their current vendor becomes unavailable. We hope that this chapter serves as an initial step toward the goal of realizing long-lasting and secure IoT devices, and ultimately toward the sustainable use of the Internet of Things.

Chapter 5.

Implementation

The previous chapter, particularly Section 4.6, lays the theoretical foundation for enhancing the durability and longevity of IoT device software. Whilst addressing concerns about the intellectual property of IoT device vendors, we proposed a compromise. Instead of forcing IoT device vendors to disclose their source code and tooling which would be a burden to open-source communities with maintaining numerous heterogeneous hardware variations, we establish a clear delineation between the vendor's responsibilities and what third-party entities can support after a transition period.

Our proposed infrastructure to support IoT longevity extends beyond the implementation of a single application or tool. To name a few of the major pieces, there is the platform-independent runtime, the implementation of vendor liveliness checks, and the integration and augmentation of cross-vendor support in existing update

frameworks, among all the small details that are needed to ensure that each of these components integrates seamlessly with the others. Implementing the full architecture is a large, multifaceted implementation effort that would extend beyond our implementation deadlines.

In this chapter, we will focus on creating a proof-of-concept implementation of the platform-independent runtime discussed in Section 4.6.1. We believe this to be a reasonable starting point as our entire design depends on the platform-independent runtime integrating with heterogeneous IoT device hardware, and additionally, several other components such as the update monitor have an inherent dependency on the existence of a platform-independent runtime.

A key demonstration of the cross-platform nature of our platform-independent runtime is execution on heterogeneous hardware. Our case study implementations use devices with varying Instruction Set Architectures (ISAs), one Advanced RISC Machines (ARM)-based, and one Reduced Instruction Set Computing (RISC)-V based. Maintainers of WebAssembly modules do not need any vendor-specific tooling or proprietary code to work with heterogeneous devices, which addresses the burdens of maintenance discussed in Section 4.6.1.

We begin by outlining the objectives for this preliminary implementation in Section 5.1. Subsequently, taking into account the outlined goals and constraints, we design and implement a WebAssembly (WASM) runtime for bare-metal embedded devices in Section 5.2. We integrate our runtime with resource-constrained embed-

ded devices commonly used in IoT in Section 5.3. Lastly, we discuss further security measures in Section 5.4 and future work in Section 5.5.

5.1. Implementation Goals

We want to illustrate how a real IoT device vendor can adopt the model presented in Chapter 4. We designed the model such that a first-party vendor does not need to re-implement all of a device's software stack from scratch: presumably the vendor already has some form of Hardware Abstraction Layer (HAL), kernel primitives, and proprietary firmware for the System on a Chips (SoCs) they build IoT devices with. The new piece a vendor would need to implement is the platform-independent runtime – a WebAssembly runtime in this case – and all the platform-independent code. In Figure 4.3, this would represent everything above and including the platform-independent runtime. This includes:

1. The runtime itself, with the following items existing as modules that run as platform-independent code.

2. Abstractions to map host peripherals and pins to more meaningful devices.

3. A peripheral access control system to ensure applications can only access certain system resources.

4. An update monitor for determining if certain installed modules are out of date, and to ensure the device is still maintained by its current vendor.

5. All applications that provide device logic and functionality, built atop the previous pieces.

For our proof of concept, we will start by leveraging a Hardware Abstraction Layer (HAL) made by the device vendor of our target boards. The HAL we are using is mainly to provide us with programmer-friendly mappings to device registers. Instead of manually setting and reading values from registers to initialize the device, configure peripheral multiplexers, and perform all input/output operations, using an existing HAL will wrap these register operations with a clean interface. Note that we are not starting with an operating system[1], everything beyond the HAL is code we write.

We chose two boards with differing hardware to demonstrate the cross-platform nature of WebAssembly. We chose an Espressif ESP32 as a commonly used IoT Microcontroller (MCU), it is based on the RISC-V architecture and includes 4MB of embedded flash and 400 KB of Static Random-Access Memory (SRAM). According to Table 2.1, the ESP32 would be categorized as a Class 3 device.

The other board we chose is the Nordic nRF52840DK, a commonly used low-power IoT development board with NFC and Bluetooth capability [88]. The nRF52840 CPU

[1]We could have started with an operating system, and then our implementation could leverage process abstractions, memory allocation, and much cleaner I/O interfaces. We chose not to use an operating system to demonstrate the feasibility of our design on embedded devices that do not leverage an existing operating system.

is based on the Arm Cortex-M4 architecture and has 1 MB flash and 256 KB of RAM. The development kit (DK) has an additional 64 MB of external SPI flash, which in our implementation we do not use. Similarly to the ESP32, the nRF52840DK is classified as a Class 3 device in Table 2.1.

5.1.1. Memory Safety

Another goal for our implementation is memory safety. Embedded systems are typically programmed in C or C++, which are languages that do not guarantee any form of memory safety, as memory is manually managed by the developer. Programs created with memory-unsafe languages are prone to bugs such as buffer overflows, use after free, null pointers, dangling pointers, and memory leaks, among other memory-related errors [107]. Requiring developers to correctly consider all the edge cases related to memory management results in bugs created out of human error.

The alternative to memory-unsafe languages is memory-safe languages, which typically come in two varieties. The most common form of memory-safe language is interpreted languages requiring some form of runtime, leaving the burden of garbage collection and other memory management tasks to the runtime itself. Languages such as Python, JavaScript, and Go fit this category. All of these languages require the additional overhead of a runtime, which involves the additional overhead of garbage collection [24].

Our implementation is written in Rust, a memory-safe systems programming lan-

guage. Unlike most memory-safe languages, Rust achieves both safety and speed without relying on runtime or a garbage collector. It accomplishes this through its extensive use of compile-time checks, which effectively identify data races and unsafe memory accesses, all without incurring any run-time overhead [78, 71]. Additionally, the HAL implementation we are using is also implemented in Rust[2], which we will discuss further in Section 5.3.

5.1.2. WebAssembly Implementation Scope

We plan on only implementing the base set of WebAssembly operation codes (op-codes) that are part of the original WebAssembly specification [98]. Thus, any extensions to the original WebAssembly core instructions that have recently been added are presently out of scope. Additionally, instructions that are part of a proposal that has not officially been accepted are out of scope. We are mainly marking these as out of scope due to time constraints, but in the future, no technical barriers are preventing us from adding on these language extensions.

5.1.3. Implementation Challenges

One of the primary challenges we will face is due to our implementation scope: a HAL only provides a unified interface to underlying device peripherals for a single

[2]At least most of the HAL implementation is in Rust. Some assembly code is included in the HAL implementations, mainly to boot the MCU.

device, but a HAL is not an operating system. Rather, an operating system can be constructed atop a HAL. The implications of this are:

- There is no standard environment. A standard environment, such as libc, provides the standard library functionality as specified by the International Organization for Standardization (ISO) C standard [61].
- There is no operating system. Without operating system primitives, we do not have memory management, thus dynamically allocating memory is not an option unless we implement a memory allocator. Likewise, other primitives offered by an operating system will not be present: no file system, no processes, and no networking stack.
- There is no scheduling or concurrency unless we implement it.

In the context of Rust, a lack of a standard environment means we are limited to the core library which is the standard set of primitive building blocks that all other Rust libraries and programs build upon [109]. The lack of a memory allocator means that dynamically-sized data structures such as vectors, hash sets, hash maps, and others will not be usable. We can use stack-allocated arrays of a fixed size and other primitive types.

Unfortunately, existing WebAssembly runtime environments do not meet our requirements. The WAMR is designed to run on resource-constrained embedded devices [33]; however, it is written in C, a language that is not memory safe [78].

Similarly, several WebAssembly runtimes have been written in Rust such as Wasm-Time [119] and Wasmer [118]; however, these runtimes depend on a standard library and allocator, making them not portable to an embedded context. As a result of the limitations of existing WebAssembly runtimes, we developed a memory-safe WebAssembly runtime that can run on embedded systems without an allocator or standard library.

5.2. Implementation of a WebAssembly Runtime

With the implementation goals and challenges established, we will now outline the implementation details of our embedded WebAssembly runtime. First, we provide some background context on the internals of WebAssembly in Section 2.5. Then, we will go over parts of our WebAssembly interpreter's implementation in Sections 5.2.1 and 5.2.2. We then use our implementation on heterogeneous embedded systems in Section 5.3.

5.2.1. Parsing WebAssembly

The WebAssembly parser takes the binary representation of a WebAssembly module and generates parts of a module instance. Note that these parts cannot be directly interpreted without first going through the compiler. The parser iterates over a WebAssembly module byte-by-byte, parsing sections, functions, and validating instruc-

tions to ultimately create an internal representation of the module for the interpreter. We present the first stages of our runtime in Figure 5.1, which we will describe below.

Our runtime is given a WebAssembly binary that is loaded into memory. The WebAssembly module is first sent through a reader which decodes the first bytes of the module and then creates a module structure that is then passed to the parser. The reader creates an iterable stream of bytes that the parser then acts upon.

The parser is responsible for extracting the various sections of a WebAssembly module, described in Section 2.5. The parser extracts the functions, function types, globals, tables, and exports. Note that at this point, any executable routines cannot be directly passed to our interpreter. At this point, we have extracted the pieces of the module, with some intermediate steps to perform.

The intermediate steps we have to perform are through a construct we have called a compiler, which is responsible for augmenting addresses inside WebAssembly modules and formatting all instructions to an executable code buffer. This is needed, as control instructions in WebAssembly do not define their targets by using a relative index[3], but instead use labels that reference outer control constructs by relative nesting depth [53, 98]. For example, a target with index 0 refers to the innermost control instruction, and larger indices move farther out. This also has the side effect of all labels being scoped: a label can only reference constructs in which they are nested [53].

[3]Relative indexes would allow us to perform in-place interpretation, where jump targets shift the instruction pointer relatively within the bytecode.

As a result of this, our compiler stage needs to resolve these relative nesting depth labels into something an interpreter can easily execute by just shifting the instruction pointer to relative positions within the module. To accomplish this, the compiler iterates all code sections, all items within that code section, and finally all items in all functions. All instructions opcodes are written to a compiled code buffer, which is used by the interpreter to directly run the WebAssembly module.

When encountering a branch instruction, the compiler determines the relative nesting depth to the target label. The relative nesting depth is calculated by finding the difference between the current nesting depth (the number of active control flow constructs on the label stack) and the nesting depth of the target label. Internally, the compiler stores the relative nesting depth and current position of the branching instruction in a *FixUp* structure, which is applied later on in the process.

After the compiler has read all instructions and written them to the compiled code buffer, we apply the *FixUp* structures. The compiled code buffer will contain all branch instructions with placeholder values, thus the *FixUp* structures are applied to resolve the placeholder values with the correct offset values that were previously calculated.

Finally, at the end of the compilation process, the compiled bytecode in the compiled code buffer is now fully resolved with all branching instructions pointing to the correct destination offsets, and the resulting WebAssembly bytecode is ready for execution.

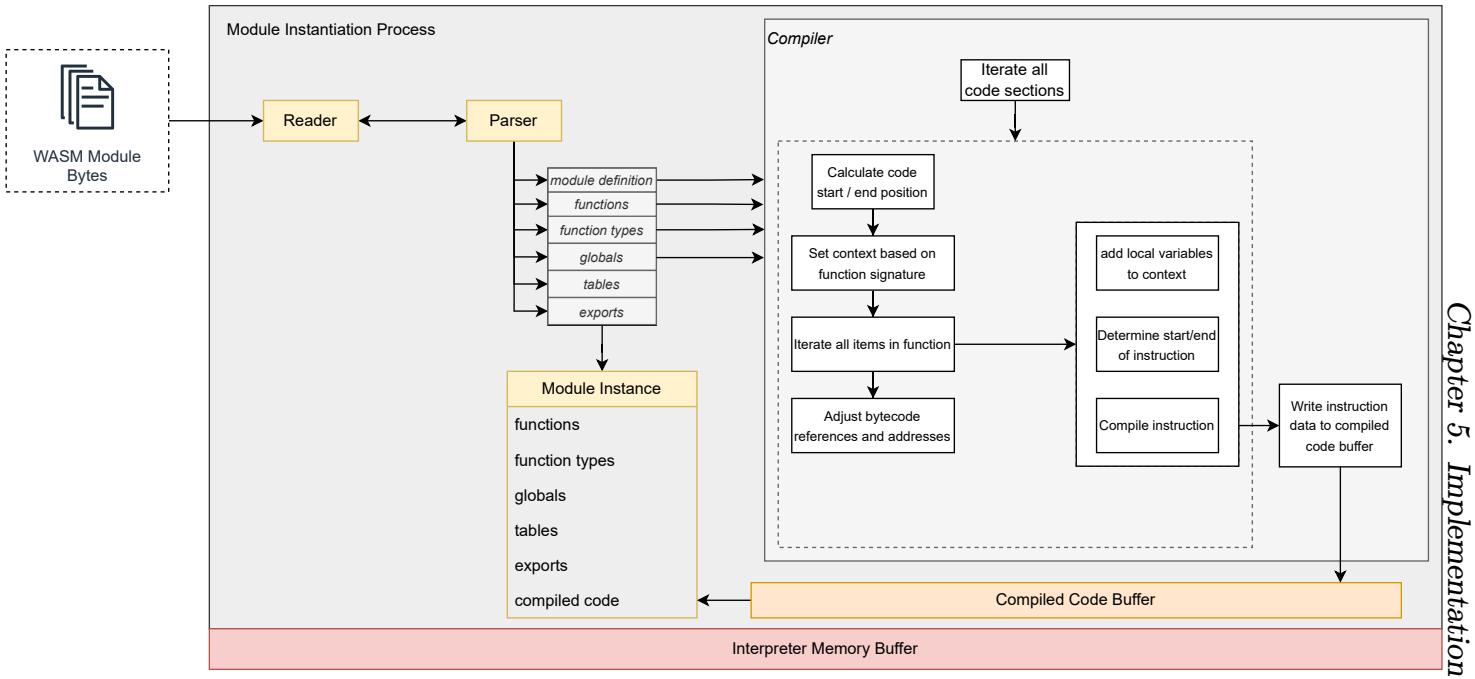


Figure 5.1.: High-level overview of module instantiation. Given a WebAssembly binary, we iterate and parse the bytes in the module into the individual components that make up the module. Executable components and dependencies get sent through a compiler stage that fixes up any internal addresses and labels, writing the resultant executable instruction to a compiled code buffer. The module instance structure can then be invoked by the interpreter.

5.2.2. Executing WebAssembly

Given an instantiated WebAssembly module, we expose an API that allows callers to invoke exported functions inside the module. For WebAssembly modules that are an entry point to IoT application logic, a module needs to export a function called `main` to begin execution. We show the high-level process in Figure 5.2, where given a function name to invoke and a module instance, the overall process of running the module is shown.

First, the interpreter needs to load the compiled code segment from the module instance. This is going to act as the instruction tape for the main interpreter loop. Using the buffer of instructions the interpreter calculates the current code position, and maximum code position, and sets up execution context structures.

The interpreter will then enter the main execution loop, as shown in Figure 5.2, this process will occur as the interpreter :

1. Read current instruction: the current opcode is read from the instruction buffer at the current position

2. Lookup opcode handler: the opcode from step 1 is parsed and a handler function is matched for the opcode.

3. The opcode handler is executed. For the sake of brevity, we will not explain how the runtime executes every opcode; however, we discuss below how certain types of opcodes are handled.

4. The opcode handler exits, in evaluating the opcode it will have accessed memory through the page table abstraction, mutated the call and/or value stacks.

5. Current position in the program is advanced

When reading opcodes, the interpreter needs to translate the opcode (an unsigned 8-bit integer) to something meaningful that it can act on. The interpreter matches all opcodes using a lookup table that contains all the standard WebAssembly opcodes [98]. This lookup table will match an opcode's number and return a structure that describes the operation in detail such that the interpreter can correctly execute the instruction.

```
pub struct Opcode {
  pub ret_type: ValueType,
  pub op1_type: ValueType,
  pub op2_type: ValueType,
  pub memory_size: u8,
  pub code: u8,
  pub text: &'static str,
}
```

Listing 5.1: WebAssembly Opcode representation as a Rust struct

The internal representation we use for an opcode is shown in Listing 5.1, which describes the metadata of what a particular opcode does. *ret_type* describes the return type of the opcode, `op1_type` and `op2_type` describe the operands to the

operation being performed, `code` is the numeric opcode, and `opcode_text` is a textual representation of the opcode for debugging purposes.

An opcode also contains a `ValueType`, which is an enum that describes what type a given value should have. At runtime, the data contained within the `Opcode` struct gives the WebAssembly interpreter enough information to act on a given instruction. We discuss how these opcode structs are used in the context of the main interpreter loop in Section 5.2.2.

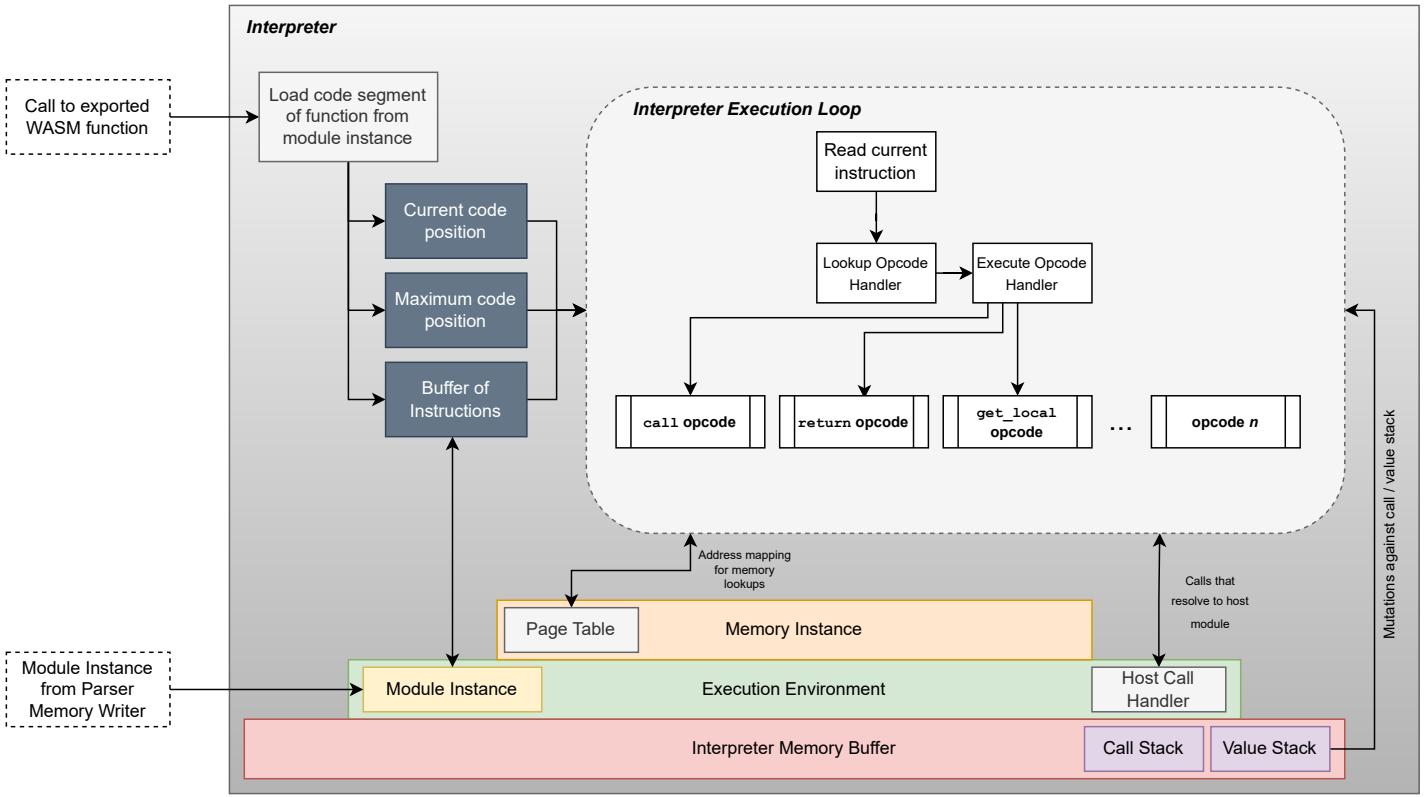


Figure 5.2.: Our WebAssembly module execution process. Given a WebAssembly module that has been instantiated, our interpreter executes the module primarily using the instructions and offsets contained in the compiled code buffer. The interpreter maintains a call and value stack for control flow and register-based operations, and we use a page table for primitive memory virtualization.

5.2.3. Memory Management

Our WebAssembly interpreter does not use a memory allocator. In fact, no part of our design makes use of dynamic memory allocation. Since our proof of concept was implemented on bare metal without an operating system we do not have an allocator available to us. On the resource-constrained embedded systems we are implementing this on, dynamic memory allocation in this context can lead to large amounts of fragmented memory, which would negatively impact performance and potentially lead to out-of-memory problems at runtime.

To maintain the high availability of a device, we opted to avoid memory allocation within the WebAssembly interpreter by giving the interpreter a fixed buffer of memory to use internally. This buffer is then managed using simple abstractions that give the illusion of dynamic memory.

The buffer is represented by a reference to a mutable array of unsigned 8-bit integers. This buffer has a size defined at compile time, thus dynamic memory allocation cannot exceed the size of the microcontroller's available memory, rather, dynamic memory allocation can only exceed the size of the statically allocated buffer. A mutable reference to the buffer is passed to the WebAssembly execution environment at the time of creation, as shown in Listing 5.2. The execution environment is created, and the reference to the mutable memory buffer is sent back for later steps.

Memory management inside the execution environment is managed using a memory instance. The memory instance structure is shown in Listing 5.3, and the memory

```
1 // Declare a mutable reference to a buffer of 16384 bytes in
     length
2 let buffer = &mut [0u8; 0x4000];
3
4 // Create execution environment using memory buffer
5 let environment = Environment::new(buffer);
```

Listing 5.2: Declaring a memory buffer as the backing memory for the WebAssembly runtime

instance creates an abstraction over the shared memory buffer. The memory instance provides several methods for loading and storing values of different types in the memory buffer.

Notably, inside the memory instance, we use a page table abstraction for managing virtual memory addresses. All load and store operations performed by a WebAssembly module go through the page table, which will map the virtual address to a real one. Note that all the memory mapping is not backed by hardware acceleration such as a MMU, thus we incur a performance penalty by emulating rudimentary MMU functionality using software.

5.3. Proof of Concept on Embedded Systems

In this section, we use our WebAssembly runtime combined with a demonstration program on real hardware. For our demonstration program, we use SPI to send commands to an Organic light-emitting diode (OLED) display to display a “hello

```
pub struct MemoryInstance<'a> {
  buf: *mut u8,
  buf_len: usize,
  page_table: UnsafeCell<PageTable>,
  num_pages: Cell<usize>,
  min_pages: usize,
  max_pages: usize
}
```

Listing 5.3: Memory instance structure

world" message. All of the application logic is implemented in WebAssembly, thus the responsibility of the host is to initialize hardware peripherals (e.g., initialize the SPI bus and system timers) and act on system calls from the WebAssembly runtime.

Our demonstration program is written in Rust and compiles down to a WebAssembly module. For the source code of the demonstration program, refer to Appendix A. An example of how we interact with host functions from a WebAssembly module is in Listing 5.5, in Rust an external function is treated as an external C function from a safety perspective.

5.3.1. Interfacing With Host Functionality

We first have to augment the WebAssembly runtime to support invoking external functions. To accomplish this, we modified the execution environment structure used by the interpreter to require an implementation of the host handler trait, pictured in

Listing 5.4. Users of the WebAssembly runtime need to provide an implementation of the import and dispatch methods, which are responsible for resolving an import from the host module, and the other for dispatching commands to host functionality.

```
1 pub trait HostHandler {
2     fn import(&self, module: &str, export: &str, import_desc:
        &ImportDesc) -> Result<usize, Error>;
3     fn dispatch(&self, interp: &mut Interpreter, mem: &
        MemoryInstance, type_index: usize, index: usize) ->
        Result<(), Error>;
4 }
```

Listing 5.4: The Rust trait that needs to be implemented for WebAssembly modules to call host functions. Import defines a function that resolves a host module import to a function, and dispatch is responsible for executing the function resolved from import.

From the perspective of WebAssembly, interfacing with host functionality is simple. WebAssembly modules perform interop via a C-style interface, instead of using Rust types and interfaces, any data needs to be passed using primitive data types or raw pointers.

On the host side, an implementation of the `HostHandler` trait is required to handle host function calls. This implementation is passed to the WebAssembly environment upon creation, such that the WebAssembly runtime can resolve and execute host functions. For the full source code of the WebAssembly program implementation, refer to Appendix A.

```rust
#[link(wasm_import_module = "host")]
extern "C" {
    fn spi_transfer(ptr: *const u8, len: i32);
    fn delay(ms: u32);
}
```

Listing 5.5: Example definitions of two host functions from a WebAssembly module written in Rust.

5.3.2. Implementation on a RISC-V Target

For a RISC-V target, we chose the ESP32-C3-DevKitM-1, which we will refer to as the ESP32 [48]. The ESP32 is a commonly used Wi-Fi-capable IoT development board. For this demonstration, we will be using the ESP32's hardware-driven SPI peripherals to connect to our OLED display.

We used an ESP HAL[4] implementation in Rust. Note that Espressif also provides ESP-IDF (ESP-Integrated Development Framework), which is a more mature development framework for ESP boards based on FreeRTOS [47]. While FreeRTOS is a microkernel-based design that does fit into our envisioned model from Section 4.6, we did not opt to use ESP-IDF and FreeRTOS due to time constraints. It would be a trivial effort to expose a C-friendly API of our runtime and integrate that into an ESP-IDF implementation. As a basic proof of concept to show input and output, a bare-metal implementation was more efficient for us to implement.

Using the ESP HAL Rust implementation, we created a simple entry point that

[4] https://github.com/esp-rs/esp-hal/

sets up system peripherals, clocks, and timers. With the basic setup completed we then set up the SPI peripheral that is compatible with the OLED display discussed in Appendix A, and a timer structure for acting on delay host calls. We created an implementation of the `HostHandler` trait that acts on all required host functions using the device peripherals.

The SPI and delay devices are ultimately structures that need to be owned by the structure that implements the `HostHandler` trait. Additionally, the ESP HAL implementation requires a mutable reference to peripherals to use them, which is a reasonable restriction to enforce memory safety.

To accomplish this, we wrapped the types in a `RefCell` which is a container type that allows us to achieve interior mutability on structure fields. Using `RefCell` comes with a runtime cost since borrow checks are performed dynamically, and a runtime panic can occur if the borrow rules are violated.

For building and loading the firmware to the device, we needed to install a Rust target[5] that will compile to the target architecture of our board. Additionally, we had to configure our Cargo[6] to use the `espflash` tool to flash the ESP32 with our firmware.

In reference to our model described in Figure 4.3, the aforementioned steps are all ones that would be performed by the first-party vendor. We were required to use

[5]Specifically, riscv32imc-unknown-none-elf.
[6]Cargo is Rust's build system and package manager.

specific compilers and tooling for this particular board, and these are ones that (per our longevity model) may not be supported indefinitely.

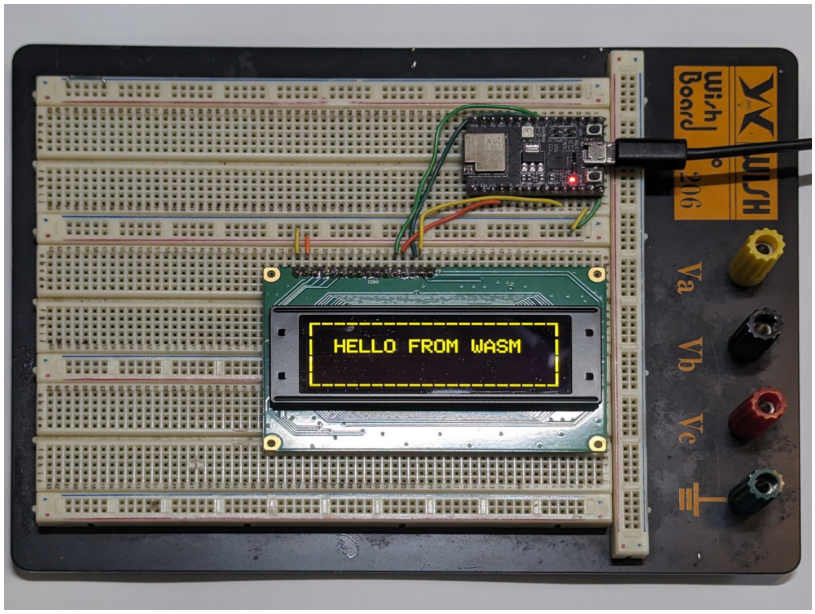
In Figure 5.3a, our "hello world" demonstration program is shown running on the ESP32. Note that all modifications to the core application logic, the "hello world" program in this case, did not involve changing the device firmware – we only had to rebuild and flash the WebAssembly demonstration program.

5.3.3. Implementation on an ARM Target

As previously mentioned we chose the Nordic Semiconductor nRF52840DK (which we also refer to as the nRF board) as our ARM-based target. Similarly to the ESP32 implementation, we found a Rust implementation of a HAL maintained by Nordic semiconductor[7]. The HAL is structured somewhat similarly to the HAL for the ESP32, only it follows slightly different vendor conventions for naming and library structure.

Similarly to the ESP implementation, we found that peripherals were structured such that mutable references are required to use them. When creating our implementation of the `HostHandler` trait, we followed a similar pattern of wrapping them in a `RefCell` for interior mutability against these instantiated peripherals. Unlike the ESP32, the nRF SPI peripheral API does not handle the Chip Select (CS) line, which is responsible for telling the SPI peripheral to wake up and send or receive data. Thus,

[7] https://github.com/nrf-rs/nrf-hal



(a) ESP32 demonstration

(b) nRF52840DK demonstration

Figure 5.3.: The "hello world" WebAssembly program running on our WebAssembly runtime using the ESP32 (top) nRF52840DK (bottom). The additional chip used on the nRF52840DK demonstration is a quad hex buffer to convert logic-level voltages.

the `HostHandler` struct on the nRF has an additional chip select General Purpose Input/Output (GPIO) pin passed through. The API difference with the chip select line has no impact on the API exposed to the WebAssembly module, it is just an additional step we need to perform when sending/receiving data over SPI.

In Figure 5.3b we show our demonstration hello world program being run using a SPI display. It is worth noting that in Figure 5.3b there is an additional chip being used compared to the ESP32 demonstration. This additional chip is required to convert the lower-voltage Transistor–Transistor Logic (TTL) logic level (2.4V) output from the nRF52840DK to a higher-level Complementary metal–oxide–semiconductor (CMOS) logic level (5V) required by the OLED display we use. Specifically, we used a 4000-series logic quad hex buffer, the 4050 [82].

The WebAssembly programs that are shown in the demonstration images are identical – they are the same program with no additional modifications made. Despite running on different hosts with different ISAs, the host peripheral API staying the same enables the operation of these demonstration programs to work despite the different underlying hardware.

5.4. Security Measures

Currently, all host peripheral calls have no security measures in place. Since we are orienting this prototype as an initial step towards the architecture presented

in Chapter 4, some control measures need to be considered to prevent potentially malicious or compromised WASM modules from compromising the device. We cannot trust all code coming from arbitrary maintainers, thus some measures need to be in place to ensure that WASM modules are only granted minimal privileges.

We suggest that alongside the WASM modules a manifest describing the module, similar to the manifests from update systems such as SUIT and TUF [81, 101]. This manifest will have an additional property of what host functions it needs access to and the actions it can perform against those peripherals. An example of such a manifest with a security policy attached is shown in Listing 5.6.

```
1  module:
2    name: Light switch application logic
3    capabilities:
4      peripherals:
5        gpio_0:
6          name: Status Indicator
7          allow: all
8        spi_0:
9          name: The flash memory chip
10         allow:
11           - transmit
12           - receive
```

Listing 5.6: Example of a manifest that describes a light switch's application logic. The peripherals map describes each named peripheral exposed to the runtime, and the given module describes what actions it can perform against system peripherals.

In Listing 5.6, lines 5 and 8 introduce named peripherals for GPIO and SPI. How-

ever, there is a concern about how WebAssembly module maintainers from outside the first-party vendor will be aware of the device's peripherals and their connections. For example, `gpio_0` might control a status indicator light on one IoT device, but on another, it could toggle the deadbolt of a smart lock. A misunderstanding of peripheral functions could lead to serious consequences.

To tackle these challenges, we propose some involvement from first-party vendors. As described in Section 4.6, first-party vendors must participate in some extent in our proposed model, which may include implementing or using a WebAssembly runtime. Additionally, device vendors should provide a publicly accessible peripheral description manifest. This manifest would serve as structured documentation, clearly describing available peripherals and host functions for the WebAssembly runtime. This way, external developers and third parties can understand and maintain specific peripheral mappings for different devices.

5.5. Discussion and Future Work

In this Chapter, we demonstrate the feasibility of utilizing a platform-independent runtime as described in Chapter 4 by building a WebAssembly runtime that targets embedded devices. This is an incremental step towards the full model described in Section 4.6.

Our prototype has several limitations, and a large amount of the future work for

this prototype revolves around addressing these limitations. While we use Rust for its attractive benefits of memory safety, we violate a lot of Rust's safety guarantees with our allocator-free model. By statically allocating a shared mutable buffer of memory, and then building the interpreter stacks, instruction buffer, execution memory, and every other component that requires some form of dynamic memory, we require the use of unsafe code. The unsafe keyword in Rust allows developers to bypass safety measures in Rust, such as preventing developers from having multiple mutable references to a piece of memory.

Solutions to deal with the usage of unsafe are to adopt an allocator that is optimized for embedded use cases, or ensure that buffers are not shared and always used by one owner. For the latter, this would involve making use of additional buffers which would store copies of data used by each component of the interpreter, thus driving up overall memory usage.

Components of our design take inspiration from the design choices made by the WebAssembly Micro Runtime (WAMR) which is written in C [33]. WAMR is a lightweight WebAssembly runtime that can run on embedded systems that provide an allocator. Other WebAssembly runtimes have been explored in IoT contexts, namely WAMR. Aerogel by Renju et al. is an access control framework designed for peripheral access in IoT devices, which is built on WAMR [76]. Another embedded runtime is WAIT by Li et al. which is a lightweight WebAssembly runtime that performs validation at compile time to reduce the additional overhead of running

WebAssembly on embedded devices [72].

A limitation of any interpreter design is performance. The additional complexity of an interpreter means that native code will always perform better, and with less complexity. While most authors of interpreters tend to argue that the relative performance of their implementation is fine for a given use case, however, for IoT these performance limitations do not appear to be significant.

5.5.1. The Future of Embedded WebAssembly

Instead of compromising and accepting the trade-off between having fast, small, native code versus the potential for code longevity via a cross-platform runtime – has poorer performance and increases the size of deployed code to IoT devices. What would a no-compromises solution look like here? The next step for this implementation is to remove the interpreter and work toward a completely native solution. Several of the limitations of our vendor agility architecture arise from the first-party vendor being the only entity capable of maintaining the microkernel and runtime. A no-compromises solution would remove the need for first-party maintenance, and achieve better performance, and smaller code sizes. The next step for this design is natively-executable WebAssembly without the need for a runtime, which would imply the CPU of these devices can natively run WebAssembly.

Perhaps this idea seems far-fetched; however, it is more possible than it may seem. Instead of using a microcontroller, a Field Programmable Gate Array (FPGA) based

solution would offer a few potential solutions to major shortcomings of the current design. As a FPGA acts as a programmable CPU, where the internal cells of the FPGA can be programmed to wire together any arbitrary CPU design, so long as the FPGA has enough resources.

Thus, a solution that has bare-metal performance would be effectively designing a WebAssembly CPU using a Hardware Definition Language (HDL) instead of a general-purpose programming language. This would allow us to create a hardware architecture designed to perform the WebAssembly parsing, compiling, and execution functions outlined in this chapter.

Running WebAssembly directly on hardware would open up new design implications for the architecture discussed in Chapter 4. As we previously discussed in Section 4.6.1, one of the primary reasons we could not open up lower levels of our IoT device architecture (Figure 4.3) is due to the inherent need for native code, and we cannot easily open up maintenance for native code on a potentially proprietary architecture with proprietary tooling. One of the compromises in the architecture was to protect the first-party vendor's intellectual property by providing a clean separation to a platform-independent runtime that contains first and n^{th} party code. Using a WebAssembly FPGA solution, an entire operating system can be created in WebAssembly. Note that for production IoT devices, a FPGA solution would be cost prohibitive. Instead, we suggest the FPGA solution as the starting point towards purpose-made hardware that can execute WebAssembly.

5.5.2. Longevity

In this chapter, we have created a proof-of-concept to demonstrate the feasibility of our envisioned architecture for extending IoT device longevity. This is only showing the proof of concept of a cross-platform runtime for embedded systems, which demonstrates that maintainers can port code to heterogeneous platforms without much effort – maintainers do not need proprietary tools and methods under our model.

Chapter 6.

Conclusion

The Internet of Things is becoming an integral part of people's daily lives, and its value is undeniable. With IoT's increasing popularity, there will be more devices manufactured, acquired, and installed in homes worldwide. However, this growth also means more effort for device vendors who must maintain the software for these devices, which is less profitable than selling new ones.

If we continue with the current state of affairs in the IoT industry, most IoT devices will become obsolete to some extent within a few years. We believe this is a result of artificially created obsolescence. Although the hardware may still function properly, outdated and vulnerable software will ultimately cause the device to fail prematurely. This not only contributes to the growing problem of e-waste on our planet [51] but also poses a threat to the security of IoT systems.

In this thesis, we raised important concerns regarding the IoT ecosystem and its

long-term software maintenance (**C1**, as discussed in Chapter 3). One major challenge is the absence of standardized software update systems, which makes it difficult for companies to create software that can be securely maintained for long periods. Additionally, continued software support for a device heavily relies on the first-party vendor, which creates a point-of-failure for the long-term security of IoT devices.

We motivated and presented a blueprint for an alternative IoT software stack specifically designed with longevity in mind (**C2**, see Chapter 4). Considering the requirements of the various stakeholders (e.g., intellectual property, proprietary tooling, allowing vendors to maintain devices for their support periods), our proposed stack enables a different party to securely take over software releases once the first-party vendor is no longer available, capable, or willing. We believe this to be a key contribution toward more long-lasting IoT devices.

We offer a proof of concept implementation of a native WebAssembly runtime (**C3**, see Chapter 5) that demonstrates how an IoT vendor can write portable code across multiple architectures, and more importantly how software can be deployed to devices even without access to the original development tooling.

In the future, we hope to see the IoT industry confront and resolve the sociotechnical challenges that arise from its operations. There are technological remedies available that can effectively mitigate these issues, and implementing them may entail some initial effort but will ultimately benefit our planet in the long term. Prioritizing the sustainability and well-being of our environment and communities is imperative,

and we believe that the IoT industry has the potential to make a significant and positive impact in this regard.

Bibliography

[1] "LineageOS Android Distribution", The LineageOS Project, 2023. URL: https://lineageos.org.

[2] "OpenSSL", OpenSSL Foundation, Inc. URL: https://www.openssl.org/.

[3] "Our decision to wind down amazon halo", Amazon, 2023. URL: https://www.aboutamazon.com/news/company-news/amazon-halo-discontinued (visited on 2023).

[4] 17 U.S Code § 1201 - Circumvention of copyright protection systems, USC, 2021.

[5] 37 CFR § 201.40 - Exemptions to prohibition against circumvention, CFR, 2011.

[6] F. J. Acosta Padilla, E. Baccelli, T. Eichinger, and K. Schleiser. The Future of IoT Software Must be Updated, 2016. URL: https://hal.inria.fr/hal-01369681.

[7] S. Albright, P. J. Leach, Y. Gu, Y. Y. Goland, and T. Cai. Simple Service Discovery Protocol/1.0. Internet-Draft, Internet Engineering Task Force, Nov. 1999. 18 pages.

[8] B. Alliance. WebAssembly System Interface, 2023. URL: https://github.com/WebAssembly/WASI.

[9] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose. Sok: security evaluation of home-based IoT deployments. In *IEEE S&P*, 2019. DOI: 10.1109/SP.2019.00013..

[10] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou. Understanding the mirai botnet:1093–1110, Aug. 2017. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis.

[11] App review - app store, Apple, Inc. URL: https://developer.apple.com/app-store/review/.

[12] Apple OTA Updates. URL: https://www.theiphonewiki.com/wiki/OTA_Updates.

[13] N. Asokan, T. Nyman, N. Rattanavipanon, A.-R. Sadeghi, and G. Tsudik. ASSURED: Architecture for Secure Software Update of Realistic Embedded

Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11), 2018. DOI: 10.1109/TCAD.2018.2858422.

[14] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt. RIOT OS: Towards an OS for the Internet of Things. In *2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 79–80, Apr. 2013. DOI: 10.1109/INFCOMW.2013.6970748.

[15] A. Banafa. Three Major Challenges Facing IoT - IEEE Internet of Things, 2017. URL: https://iot.ieee.org/newsletter/march-2017/three-major-challenges-facing-iot.html/.

[16] W. Barker. Getting Ready for Post-Quantum Cryptography: Exploring Challenges Associated with Adopting and Using Post-Quantum Cryptographic Algorithms. Technical report, 2021. DOI: 10.6028/nist.cswp.15.

[17] D. Barrera, D. McCarney, J. Clark, and P. C. van Oorschot. Baton: certificate agility for android's decentralized signing infrastructure. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*. Association for Computing Machinery, 2014. DOI: 10.1145/2627393.2627397.

[18] D. Barrera and P. Van Oorschot. Secure software installation on smartphones. *IEEE Security & Privacy*, 9(3), 2010. DOI: 10.1109/MSP.2010.202..

[19] J. Bauwens, P. Ruckebusch, S. Giannoulis, I. Moerman, and E. D. Poorter. Over-the-Air Software Updates in the Internet of Things: An Overview of

Key Principles. *IEEE Communications Magazine*, 58(2), 2020. DOI: 10.1109/MCOM.001.1900125.

[20] A. Bellissimo, J. Burgess, and K. Fu. Secure software updates: disappointments and new challenges. In *First USENIX Workshop on Hot Topics in Security*, July 2006.

[21] C. Bellman and P. C. Van Oorschot. Analysis, implications, and challenges of an evolving consumer iot security landscape. In *2019 17th International Conference on Privacy, Security and Trust*. IEEE, 2019. DOI: 10.1109/PST47121.2019.8949058..

[22] C. Bellman and P. C. van Oorschot. Analysis, Implications, and Challenges of an Evolving Consumer IoT Security Landscape. en. In *17th International Conference on Privacy, Security and Trust (PST)*, pages 1–7, Fredericton, NB, Canada. IEEE, Aug. 2019. ISBN: 978-1-72813-265-5. DOI: 10.1109/PST47121.2019.8949058.

[23] M. Bettayeb, Q. Nasir, and M. A. Talib. Firmware update attacks and security for iot devices: survey. In *Proceedings of the ArabWIC 6th Annual International Conference Research Track*, ArabWIC, Rabat, Morocco. Association for Computing Machinery, 2019. ISBN: 9781450360890. DOI: 10.1145/3333165.3333169.

[24] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: the performance impact of garbage collection. In *Proceedings of the Joint Interna-*

tional Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '04/Performance '04, pages 25–36, New York, NY, USA. Association for Computing Machinery, 2004. ISBN: 1581138733. DOI: 10.1145/1005686.1005693.

[25] C. Bormann, M. Ersue, and A. Keränen. Terminology for Constrained-Node Networks. RFC 7228, 2014. DOI: 10.17487/RFC7228.

[26] C. Bormann, M. Ersue, A. Keränen, and C. Gomez. Terminology for Constrained-Node Networks. Internet-Draft draft-ietf-lwig-7228bis-00, Internet Engineering Task Force, June 2022. 27 pages. URL: https://datatracker.ietf.org/doc/draft-ietf-lwig-7228bis/00/. Work in Progress.

[27] A. Boudguiga, N. Bouzerna, L. Granboulan, A. Olivereau, F. Quesnel, A. Roger, and R. Sirdey. Towards better availability and accountability for IoT updates by means of a blockchain. In *IEEE Euro S&PW*, 2017. DOI: 10.1109/EuroSPW.2017.50.

[28] C. Bradley and D. Barrera. Escaping Vendor Mortality: A New Paradigm for Extending IoT Device Longevity. To appear in *proceedings of the 2023 New Security Paradigms Workshop*, NSPW '23, Segovia, Spain. Association for Computing Machinery, 2023.

[29] C. Bradley and D. Barrera. Towards characterizing IoT software update practices. In *Foundations and Practice of Security*, pages 406–422. Springer, 2023. DOI: 10.1007/978-3-031-30122-3_25.

[30] P. Break Free From. Brand audit report 2021, 2021. URL: https://brandaudit.breakfreefromplastic.org/brand-audit-2021/.

[31] T. Brydges. Closing the loop on take, make, waste: investigating circular economy practices in the swedish fashion industry. *Journal of Cleaner Production*, 293, 2021. DOI: 10.1016/j.jclepro.2021.126245.

[32] D. Buentello. Belkin wemo - arbitrary firmware upload, Apr. 2013. URL: https://www.exploit-db.com/exploits/24924.

[33] WebAssembly Micro Runtime (WAMR), The ByteCode Alliance, 2022. URL: https://github.com/bytecodealliance/wasm-micro-runtime.

[34] R. Chen. If you configure a program to run in windows 2000 compatibility mode, then it is also vulnerable to windows 2000 security issues, Mar. 2019. URL: https://devblogs.microsoft.com/oldnewthing/20170911-00/?p=96995.

[35] R. Chen. Why not just block the apps that rely on undocumented behavior?, Dec. 2003. URL: https://devblogs.microsoft.com/oldnewthing/20031224-00/?p=41363.

[36] E. Chung. Companies can - and may - brick your connected devices at any time, Apr. 2016. URL: https://www.cbc.ca/news/science/revolv-bricked-1.3521927.

[37] T. Cooper. Beyond recycling: the longer life option, 1994.

[38] T. Cooper. Inadequate life? evidence of consumer attitudes to product obsolescence. *Journal of Consumer Policy*, 27(4), 2004. DOI: 10.1007/s10603-004-2284-6.

[39] T. Cooper. *The significance of product longevity.* In. *Longer Lasting Products.* Routledge, 2016, pages 3–36.

[40] A. Cui, M. Costello, and S. Stolfo. When firmware modifications attack: a case study of embedded exploitation. *NDSS*, 2013. DOI: https://doi.org/10.7916/D8P55NKB.

[41] CVE-2008-4395, National Vulnerability Database. URL: https://nvd.nist.gov/vuln/detail/CVE-2008-4395.

[42] D. DiClerico. Hp inkjet printer lawsuit reaches $5 million settlement, 2010. URL: https://www.consumerreports.org/cro/news/2010/11/hp-inkjet-printer-lawsuit-reaches-5-million-settlement/index.htm.

[43] E. Dils, J. Bachér, Y. Dams, T. Duhoux, Y. Deng, and T. Teittinen. Electronics and obsolescence in a circular economy, 2020. URL: https://www.eionet.europa.eu/etcs/etc-wmge/products/etc-wmge-reports/electronics-and-obsolescence-in-a-circular-economy.

[44] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462, Nov. 2004. DOI: 10.1109/LCN.2004.38.

[45] S. El Jaouhari and E. Bouvet. Secure firmware Over-The-Air updates for IoT: Survey, challenges, and discussions. *Internet of Things*, 18, 2022. DOI: 10.1016/j.iot.2022.100508.

[46] End of service and instructions for a standalone option, Io, Monitor, 2023. URL: https://www.monitor-io.com/.

[47] ESP-IDF: FreeRTOS (Overview). URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/freertos.html.

[48] ESP32-C3-DevKitM-1. URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32c3/hw-reference/esp32c3/user-guide-devkitm-1.html.

[49] European Commission. Directorate General for the Environment. and Ricardo AEA Ltd. *The durability of products: standard assessment for the circular economy under the eco innovation action plan.* Publications Office, 2015. URL: https://data.europa.eu/doi/10.2779/37050.

[50] Bill C-244 441 An Act to amend the Copyright Act (diagnosis, maintenance and repair), Federal Trade Commission, 2023.

[51] V. Forti, C. P. Baldé, R. Kuehr, and G. Bel. The global e-waste monitor 2020. *Quantities, flows, and the circular economy potential*, 2020.

[52] S. Gibbs. Is the year 2038 problem the new y2k bug?, Dec. 2014. URL: https://www.theguardian.com/technology/2014/dec/17/is-the-year-2038-problem-the-new-y2k-bug.

[53] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 185–200, Barcelona, Spain. Association for Computing Machinery, 2017. ISBN: 9781450349888. DOI: 10.1145/3062341.3062363.

[54] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes. Operating Systems for Low-End Devices in the Internet of Things: A Survey. *IEEE Internet of Things Journal*, 3(5):720–734, Oct. 2016. ISSN: 2327-4662. DOI: 10.1109/JIOT.2015.2505901.

[55] X. He, S. Alqahtani, R. Gamble, and M. Papa. Securing over-the-air iot firmware updates using blockchain. In *Proceedings of the International Conference on Omni-Layer Intelligent Systems*, COINS '19, pages 164–171, Crete, Greece. Association for Computing Machinery, 2019. ISBN: 9781450366403. DOI: 10.1145/3312614.3312649.

[56] J. L. Hernández-Ramos, G. Baldini, S. N. Matheu, and A. Skarmeta. Updating IoT devices: challenges and potential approaches. In *2020 Global Internet of Things Summit (GIoTS)*. IEEE, 2020. DOI: 10.1109/GIOTS49054.2020.9119514.

[57] S. Higginbotham. The internet of trash [internet of everything]. *IEEE Spectrum*, 55, 2018. DOI: 10.1109/MSPEC.2018.8362218.

[58] S. Higginbotham. The iot's e-waste problem isn't inevitable, July 2021. URL: https://spectrum.ieee.org/the-iots-ewaste-problem-isnt-inevitable.

[59] J. Hsu. Why the military can't quit windows xp, June 2018. URL: https://slate.com/technology/2018/06/why-the-military-cant-quit-windows-xp.html.

[60] M. Ibrahim, A. Continella, and A. Bianchi. Aot - attack on things: a security analysis of iot firmware updates. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, July 2023.

[61] Information technology – Programming languages – C. Standard, International Organization for Standardization, Geneva, CH, 2018.

[62] S. Jiang, K. A. Britt, A. J. McCaskey, T. S. Humble, and S. Kais. Quantum annealing for prime factorization. *Scientific Reports*, 8(1), 2018. DOI: 10.1038/s41598-018-36058-z.

[63] F. Jindal, R. Jamar, and P. Churi. Future and challenges of internet of things. *Int. J. Comput. Sci. Inf. Technol*, 10(2):13–25, 2018. DOI: 10.5121/ijcsit.2018.10202.

[64] T. Karthik, A. Brown, S. Awwad, D. McCoy, R. Bielawski, C. Mott, S. Lauzon, A. Weimerskirch, and J. Cappos. Uptane: securing software updates for automobiles. In *The 14th escar Europe*, 2016.

[65] M. Khurana, T. P. Singh, and T. Choudhury. Effective threat and security modelling approach to devise security rating of diverse IoT devices. In *Data Driven Approach Towards Disruptive Technologies*, pages 583–593. Springer Singapore, 2021. DOI: 10.1007/978-981-15-9873-9_46.

[66] K. Kiningham, M. Horowitz, P. Levis, and D. Boneh. Cesel: securing a mote for 20 years. In *Proceedings of the 2016 International Conference on Embedded Wireless Systems and Networks*, EWSN '16, pages 307–312, Graz, Austria. Junction Publishing, 2016. ISBN: 9780994988607.

[67] G. Klein, M. Norrish, T. Sewell, H. Tuch, S. Winwood, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, and R. Kolanski. seL4: formal verification of an OS kernel. en. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles - SOSP '09*, page 207, Big Sky, Montana, USA. Association for Computing Machinery, 2009. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596.

[68] A. Langiu, C. A. Boano, M. Schuss, and K. Romer. UpKit: an open-source, portable, and lightweight update framework for constrained IoT devices. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, July 2019. DOI: 10.1109/icdcs.2019.00207.

[69] O. Leiba, Y. Yitzchak, R. Bitton, A. Nadler, and A. Shabtai. Incentivized delivery network of iot software updates based on trustless proof-of-distribution. In *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 29–39, 2018. DOI: 10.1109/EuroSPW.2018.00011.

[70] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Pannuto, P. Dutta, and P. Levis. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*. Association for Computing Machinery, 2017. DOI: 10.1145/3132747.3132786.

[71] A. Levy, B. Campbell, B. Ghena, P. Pannuto, P. Dutta, and P. Levis. The case for writing a kernel in rust. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, APSys '17, Mumbai, India. Association for Computing Machinery, 2017. ISBN: 9781450351973. DOI: 10.1145/3124680.3124717.

[72] B. Li, H. Fan, Y. Gao, and W. Dong. Bringing webassembly to resource-constrained iot devices for seamless device-cloud integration. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, MobiSys '22, pages 261–272, Portland, Oregon. Association for

Computing Machinery, 2022. ISBN: 9781450391856. DOI: 10.1145/3498361.3538922.

[73] J. Liedtke. On micro-kernel construction. *ACM SIGOPS Operating Systems Review*, 29(5):237–250, 1995.

[74] Lightweight machine to machine technical specification: core, 2019. URL: https://www.openmobilealliance.org/release/LightweightM2M/V1_1_1-20190617-A/OMA-TS-LightweightM2M_Core-V1_1_1-20190617-A.pdf.

[75] Linux 2.6.31 rc3: security vulnerabilities, CVE Details. URL: https://www.cvedetails.com/version/446073/Linux-Linux-Kernel-2.6.31-rc3.html.

[76] R. Liu, L. Garcia, and M. Srivastava. Aerogel: Lightweight Access Control Framework for WebAssembly-Based Bare-Metal IoT Devices. In *2021 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2021. DOI: 10.1145/3453142.3491282.

[77] N. Mäkitalo, T. Mikkonen, C. Pautasso, V. Bankowski, P. Daubaris, R. Mikkola, and O. Beletski. WebAssembly modules as lightweight containers for liquid IoT applications. In *Lecture Notes in Computer Science*, pages 328–336. Springer International Publishing, 2021. DOI: 10.1007/978-3-030-74296-6_25.

[78] N. D. Matsakis and F. S. Klock. The rust language. 34(3), 2014. ISSN: 1094-3641. DOI: 10.1145/2692956.2663188.

[79] B. Moran, Ø. Rønningstad, and A. Tsukamoto. Mandatory-to-Implement Algorithms for Authors and Recipients of Software Update for the Internet of Things manifests. Internet-Draft draft-ietf-suit-mti-01, Internet Engineering Task Force, July 2023. 10 pages. URL: https://datatracker.ietf.org/doc/draft-ietf-suit-mti/01/. Work in Progress.

[80] B. Moran, H. Tschofenig, and H. Birkholz. A Manifest Information Model for Firmware Updates in Internet of Things (IoT) Devices. RFC 9124, Jan. 2022.

[81] B. Moran, H. Tschofenig, D. Brown, and M. Meriac. A Firmware Update Architecture for Internet of Things. Request for Comments RFC 9019, Internet Engineering Task Force, 2021. DOI: 10.17487/RFC9019.

[82] *CMOS Integrated Circuits Databook*. Third printing. Motorola Inc. 1978, pages 7–129 – 7–132.

[83] P.-E. Moyse. The uneasy case of programmed obsolescence. *UNB Law Journal*, 71:61, 2020.

[84] S. Murakami, M. Oguchi, T. Tasaki, I. Daigo, and S. Hashimoto. Lifespan of Commodities, Part I. *Journal of Industrial Ecology*, 14(4), 2010. DOI: 10.1111/j.1530-9290.2010.00250.x.

[85] Network Time Protocol (Version 3) Specification, Implementation and Analysis. RFC 1305, Mar. 1992. DOI: 10.17487/RFC1305.

[86] K. Nikitin, E. Kokoris-Kogias, P. Jovanovic, N. Gailly, L. Gasser, I. Khoffi, J. Cappos, and B. Ford. CHAINIAC: proactive Software-Update transparency via collectively signed skipchains and verified builds. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1271–1287, Vancouver, BC. USENIX Association, Aug. 2017. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/nikitin.

[87] Nixing the Fix: An FTC Report to Congress on Repair Restrictions, Federal Trade Commission, 2021.

[88] nRF52840 DK, Nordic Semiconductor, 2021. URL: https://infocenter.nordicsemi.com/pdf/nRF52840_PS_v1.7.pdf.

[89] T. OConnor, W. Enck, and B. Reaves. Blinded and confused: uncovering systemic flaws in device telemetry for smart-home internet of things. In *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '19, pages 140–150, Miami, Florida. Association for Computing Machinery, 2019. ISBN: 9781450367264. DOI: 10.1145/3317549.3319724.

[90] M. O. Ojo, S. Giordano, G. Procissi, and I. N. Seitanidis. A review of low-end, middle-end, and high-end iot devices. *IEEE Access*, 6:70528–70554, 2018. DOI: 10.1109/ACCESS.2018.2879615.

[91] M. T. Paracha, D. J. Dubois, N. Vallina-Rodriguez, and D. Choffnes. Iotls: understanding tls usage in consumer iot devices. In *Proceedings of the 21st ACM*

Internet Measurement Conference, IMC '21, pages 165–178, Virtual Event. Association for Computing Machinery, 2021. ISBN: 9781450391290. DOI: 10.1145/3487552.3487830.

[92] J. Peters. Google discontinues its google nest secure alarm system, 2020. URL: https://www.theverge.com/2020/10/19/21523967/google-discontinues-nest-secure-alarm-system.

[93] M. Pizzetti, L. Gatti, and P. Seele. Firms talk, suppliers walk: analyzing the locus of greenwashing in the blame game and introducing 'vicarious greenwashing'. *Journal of Business Ethics*, 170(1), 2019. DOI: 10.1007/s10551-019-04406-2.

[94] V. Prakash, S. Xie, and D. Y. Huang. Software Update Practices on Smart Home IoT Devices. en, Sept. 2022. URL: http://arxiv.org/abs/2208.14367 (visited on 10/04/2022).

[95] Qualys SSL Labs, Qualys Inc. URL: https://www.ssllabs.com/.

[96] L. F. Rahman, T. Ozcelebi, and J. Lukkien. Understanding IoT systems: a life cycle approach. *Procedia Computer Science*, 2018. ISSN: 1877-0509. DOI: 10.1016/j.procs.2018.04.148.

[97] J. Ren, D. J. Dubois, D. Choffnes, A. M. Mandalari, R. Kolcun, and H. Haddadi. Information exposure from consumer iot devices: a multidimensional, network-informed measurement approach. In *Proceedings of the Internet Mea-*

surement Conference, IMC '19, pages 267–279, Amsterdam, Netherlands. Association for Computing Machinery, 2019. ISBN: 9781450369480. DOI: 10.1145/3355369.3355577.

[98] A. Rossberg, editor. Webassembly specification release 2.0, 2023. URL: https://webassembly.github.io/spec/core/_download/WebAssembly.pdf.

[99] H. C. Rudolph and N. Grundmann. TLS ciphersuite search, Ciphersuite Info. URL: https://ciphersuite.info/.

[100] N. Sajn. Briefing: Right to Repair, 2022. URL: https://www.europarl.europa.eu/RegData/etudes/BRIE/2022/698869/EPRS_BRI(2022)698869_EN.pdf.

[101] J. Samuel, N. Mathewson, J. Cappos, and R. Dingledine. Survivable key compromise in software update systems. In *17th ACM CCS*. Association for Computing Machinery, 2010. DOI: 10.1145/1866307.1866315.

[102] Secure software updates, Apple Inc., 2021. URL: https://support.apple.com/en-ca/guide/security/secf683e0b36/web.

[103] M. R. Shahid, G. Blanc, Z. Zhang, and H. Debar. Iot devices recognition through network traffic analysis. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 5187–5192, 2018. DOI: 10.1109/BigData.2018.8622243.

[104] R. W. Shirey. Internet Security Glossary, Version 2. RFC 4949, Aug. 2007. DOI: 10.17487/RFC4949.

[105] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts, Seventh Edition*. John Wiley & Sons, Dec. 2004, page 921. ISBN: 978-0471694663.

[106] M. R. Stead, P. Coulton, J. G. Lindley, and C. Coulton. The little book of sustainability for the internet of things, 2019.

[107] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62, 2013. DOI: 10.1109/SP.2013.13.

[108] A. S. Tanenbaum and H. Bos. *Modern Operating Systems*. Pearson, Upper Saddle River, NJ, 4th edition, Mar. 2014. ISBN: 978-0133591620.

[109] The Rust Core Library, 2023. URL: https://doc.rust-lang.org/core.

[110] L. Torvalds. Linux 2.6.31 released, Sept. 2009. URL: https://www.linux.com/news/linux-2631-released/.

[111] L. Torvalds. Linux kernel mailing list: media commit causes user space to misbahave, 2012. URL: https://lkml.org/lkml/2012/12/23/75 (visited on 2023).

[112] D. Tracey and C. Sreenan. OMA LWM2m in a holistic architecture for the internet of things. In *2017 IEEE 14th International Conference on Networking, Sensing and Control (ICNSC)*. IEEE, 2017. DOI: 10.1109/icnsc.2017.8000091.

[113] H. Tschofenig and S. Farrell. Report from the Internet of Things Software Update (IoTSU) Workshop 2016. RFC 8240, 2017. DOI: 10.17487/RFC8240.

[114] H. Tschofenig, R. Housley, and B. Moran. Firmware Encryption with SUIT Manifests. Internet-Draft, Internet Engineering Task Force, Oct. 2021. 19 pages.

[115] S. Vasile, D. Oswald, and T. Chothia. Breaking All the Things—A Systematic Survey of Firmware Extraction Techniques for IoT Devices. In *Smart Card Research and Advanced Applications*, Lecture Notes in Computer Science. Springer, 2019. DOI: 10.1007/978-3-030-15462-2_12.

[116] S. Vaughan-Nichols. No ink, no scan: canon usa printers hit with class-action suit, 2021. URL: https://www.zdnet.com/article/untrustworthy-canon-printer-lawsuit/.

[117] A. Wang, R. Liang, X. Liu, Y. Zhang, K. Chen, and J. Li. An inside look at IoT malware. In *Industrial IoT Technologies and Applications*, LNICST. Springer, 2017. DOI: 10.1007/978-3-319-60753-5_19.

[118] Wasmer, 2023. URL: https://wasmer.io/.

[119] Wasmtime, 2023. URL: https://wasmtime.dev/.

[120] E. Wen and G. Weber. Wasmachine: bring iot up to speed with a webassembly os. In *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 1–4, 2020. DOI: 10.1109/PerComWorkshops48775.2020.9156135.

[121] G. Wurster and P. C. van Oorschot. The developer is the enemy. In *Proceedings of the 2008 New Security Paradigms Workshop*, NSPW '08, pages 89–97. Association for Computing Machinery, 2008. ISBN: 9781605583419. DOI: 10.1145/1595676.1595691.

[122] J.-Y. Yu and Y.-G. Kim. Analysis of IoT platform security: a survey. In *International Conference on Platform Technology and Service (PlatCon)*, 2019. DOI: 10.1109/PlatCon.2019.8669423.

[123] K. Zandberg, K. Schleiser, F. Acosta, H. Tschofenig, and E. Baccelli. Secure firmware updates for constrained iot devices using open standards: a reality check. *IEEE Access*, 7:71907–71920, 2019. DOI: 10.1109/ACCESS.2019.2919760..

[124] P. Zdankin and T. Weis. Longevity of smart homes. In *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 1–2, 2020. DOI: 10.1109/PerComWorkshops48775.2020.9156155.

[125] H. Zhang, A. Anilkumar, M. Fredrikson, and Y. Agarwal. Capture: centralized library management for heterogeneous IoT devices. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 4187–4204. USENIX Association, Aug. 2021. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/zhang-han.

Appendix A.

WebAssembly Demo Program

This demo program[1] uses SPI on the host to interact with an OLED display. The specific display we used is a Newhaven NHD-0420DZW-AY5. Note that to this particular display will not work with SPI by default, it instead uses a 6800 parallel interface. Several jumpers on the back of the display need to be de-soldered and re-soldered to short specific pads, refer to the "Jumper Selections" table of the datasheet.

To ensure the WebAssembly binary that is emitted by this listing is small enough to fit in an embedded device's memory, ensure that the binary is built as follows:

- Build as a release build to ensure that debug symbols are not included

  ```
  cargo build -target=wasm32-unknown-unknown -release
  ```

- Optionally use the wasm-opt tool to ensure the sign extension is not part of the

[1] Available for download at
https://www.cisl.carleton.ca/~cbradley/data/snippets/wasm_spi_demo.rs

resulting wasm binary

```
wasm-opt --signext-lowering \
  -Os target/wasm32-unknown-unknown/release/demo\_wasm.wasm \
  -o demo\_wasm.wasm
```

```
// External functions implemented by the host
#[link(wasm_import_module = "host")]
extern "C" {
    // Transfers data via a SPI peripheral initalized by the host
    fn spi_transfer(ptr: *const u8, len: i32);

    // Performs a delay
    fn delay(ms: u32);
}

struct Spi {}
impl Spi {
    fn transfer(&mut self, words: &mut [u8]) -> Result<&[u8], ()> {
        unsafe {
            spi_transfer(words.as_ptr(), words.len() as i32);
        }
        Ok(&[0 as u8])
    }
}
// Writes a "command" instruction to the SPI display with a payload
fn write_command(spi: &mut Spi, cmd: u8) {
    let mut packet = [0; 2];

```

```
24      // Specify a command instruction
25      // RS = 0, RW = 0
26      packet[0] = 0b0000_0000;
27
28      // Set the command bits
29      packet[0] |= (cmd & 0b1111_1100) >> 2;
30      packet[1] |= (cmd & 0b0000_0011) << 6;
31
32      // Transfer the command via the peripheral
33      spi.transfer(&mut packet).unwrap();
34  }
35
36  // Writes a "data" instruction to the SPI display witih a payload
37  fn write_data(spi: &mut Spi, data: u8) {
38      let mut packet = [0; 2];
39
40      // Specify data instruction
41      // RS = 1, RW = 0
42      packet[0] = 0b1000_0000;
43
44      // Set the data bits
45      packet[0] |= (data & 0b1111_1100) >> 2;
46      packet[1] |= (data & 0b0000_0011) << 6;
47
48      spi.transfer(&mut packet).unwrap();
49  }
50
51  // Moves to a specific position on the SPI display
52  fn move_to_command(x: u8, y: u8) -> u8 {
53      // Corresponds to the first index of each row of the 4 row display
54      // derived from datasheet
55      let row_address = match y {
```

```
56          0 => 0x00,
57          1 => 0x40,
58          2 => 0x14,
59          3 => 0x54,
60          _ => 0x00,
61      };
62      let mut position_address = row_address + x;
63
64      // Datasheet: the position change command has db7 high
65      // bits from db6-db0 are the address
66      position_address |= 0b1000_0000;
67      position_address
68  }
69
70  #[no_mangle]
71  pub fn main() {
72      // Create SPI peripheral (with enforced ownership)
73      let mut spi = Spi {};
74
75      // Perform display initalization sequence
76      write_command(&mut spi, 0b0011_1011);   //function set
77      write_command(&mut spi, 0x06);          //entry mode set
78      write_command(&mut spi, 0x02);          //return home
79      write_command(&mut spi, 0x01);          //clear display
80      write_command(&mut spi, 0x0c);          //display on
81      write_command(&mut spi, 0x80);          //line 1 character 1
82
83      // CGRAM is stored in positions 0x00 through 0x07 of the font table.
84      // Therefore, to write the first CGRAM character to the display, you would
85      // move the cursor to the desired DDRAM location on the display and write
            character data 0x00.
86
```

```
87      // Set CGRAM Address in address counter to 0x00
88      write_command(&mut spi, 0x40);
89      // Create custom character (vertical line on left edge for box drawing)
90      write_data(&mut spi, 0b0001_0000);
91      write_data(&mut spi, 0b0001_0000);
92      write_data(&mut spi, 0b0001_0000);
93      write_data(&mut spi, 0b0001_0000);
94      write_data(&mut spi, 0b0001_0000);
95      write_data(&mut spi, 0b0001_0000);
96      write_data(&mut spi, 0b0001_0000);
97      write_data(&mut spi, 0b0001_0000);
98      let box_draw_bar_left: u8 = 0x00;   // cgram 0
99
100     write_command(&mut spi, 0x48);
101     // Create custom character (vertical line on right edge for box drawing)
102     write_data(&mut spi, 0b0000_0001);
103     write_data(&mut spi, 0b0000_0001);
104     write_data(&mut spi, 0b0000_0001);
105     write_data(&mut spi, 0b0000_0001);
106     write_data(&mut spi, 0b0000_0001);
107     write_data(&mut spi, 0b0000_0001);
108     write_data(&mut spi, 0b0000_0001);
109     write_data(&mut spi, 0b0000_0001);
110     let box_draw_bar_right: u8 = 0x01;  // cgram 1
111
112     // Other box drawing characters (from font table )
113     let box_draw_upper_left: u8 = 0b1100_1001;
114     let box_draw_upper_right: u8 = 0b1100_1010;
115     let box_draw_lower_right: u8 = 0b1100_1100;
116     let box_draw_lower_left: u8 = 0b1100_1011;
117
118     let box_draw_upper_line: u8 = 0b1011_1111;
```

```
119     let box_draw_lower_line: u8 = 0b0101_1111;
120
121     let message = "HELLO FROM WASM";
122     let mut msg_position = 1;
123
124     write_command(&mut spi, 0b0000_0001);
125
126     // Draw upper box
127     write_command(&mut spi, move_to_command(0, 0));
128     write_data(&mut spi, box_draw_upper_left);
129     for _i in 0..18 {
130         write_data(&mut spi, box_draw_upper_line);
131     }
132     write_data(&mut spi, box_draw_upper_right);
133
134     // Draw lower box
135     write_command(&mut spi, move_to_command(0, 3));
136     write_data(&mut spi, box_draw_lower_left);
137
138     for _i in 0..18 {
139         write_data(&mut spi, box_draw_lower_line);
140     }
141     write_data(&mut spi, box_draw_lower_right);
142
143     // Left box edge
144     write_command(&mut spi, move_to_command(0, 1));
145     write_data(&mut spi, box_draw_bar_left);
146     write_command(&mut spi, move_to_command(0, 2));
147     write_data(&mut spi, box_draw_bar_left);
148
149     // Right box edge
150     write_command(&mut spi, move_to_command(19, 1));
```

```
151     write_data(&mut spi, box_draw_bar_right);
152     write_command(&mut spi, move_to_command(19, 2));
153     write_data(&mut spi, box_draw_bar_right);
154
155     // Box drawn, begin main loop:
156     loop {
157         // Clear display
158         if (msg_position + message.len()) > (18 + message.len()) {
159             msg_position = (0 - message.len()) + 2;
160
161             for i in 1..19 {
162                 write_command(&mut spi, move_to_command(i, 1));
163                 write_data(&mut spi, ' ' as u8);
164             }
165         }
166
167         // Draw data inside the box
168         // Clear row\
169         write_command(&mut spi, move_to_command(1, 1));
170         let mut i: usize = 0;
171         if msg_position > 1 {
172             write_command(&mut spi, move_to_command((msg_position - 1) as u8, 1));
173             write_data(&mut spi, ' ' as u8);
174         }
175
176         for char in message.chars() {
177             // Calculate the position of the current character on the screen
178             let position = i + msg_position;
179
180             // Is the character in bounds?
181             if position > 0 && position <= 18 {
182                 if i == 0 {
```

```
183                     write_command(&mut spi, move_to_command(position as u8, 1));
184                 }
185
186                 write_data(&mut spi, char as u8);
187             }
188
189             i += 1;
190         }
191
192         write_command(&mut spi, 0x80);
193         msg_position += 1;
194
195         // Pause before next loop
196         unsafe {
197             delay(500);
198         }
199     }
200 }
```

Listing A.1: WebAssembly SPI demonstration program

List of Abbreviations

ABI Application Binary Interface. 80

API Application Programming Interface. 48, 67, 79, 86, 101, 106, 115, 151, 155

ARM Advanced RISC Machines. 133

ASSURED Architecture for Secure Software Update of Realistic Embedded Devices. 32

BIOS Basic Input/Output System. 85

BLE Bluetooth Low Energy. 20

CA Certificate Authority. 106, 107

CDN Content Delivery Network. 66, 67

CMOS Complementary metal–oxide–semiconductor. 155

CoAP Constrained Application Protocol. 25

CPU Central Processing Unit. 13, 63, 135, 160

CS Chip Select. 153

CVE Common Vulnerabilities and Exposures. 52

DB Database. 46

DNS Domain Name System. 50, 118

DTLS Datagram Transport Layer Security. 25

ECID Electronic Control Identification. 68

ECU Engine Control Unit. 24

FPGA Field Programmable Gate Array. 159, 160

FTC Federal Trade Commission. 125

GB Gigabyte. 43, 46

GPIO General Purpose Input/Output. 155, 156

HAL Hardware Abstraction Layer. 111, 114, 134, 135, 137, 138

HDL Hardware Definition Language. 160

HTTP Hypertext Transfer Protocol. 41, 42, 44–46, 49–52, 55, 56, 60–62, 65, 67, 69, 70

HTTPS Hypertext Transfer Protocol Secure. 68

IANA Internet Assigned Numbers Authority. 48

IETF Internet Engineering Task Force. x, 13, 23, 31–33, 71, 79, 81, 91

IoT Internet of Things. i, ii, vi, x, 2–9, 11–17, 20, 22–26, 29, 30, 32–34, 38–44, 47–50, 52, 54, 55, 57, 58, 60, 61, 63, 65, 70–72, 74–134, 157, 160, 162, 163

IP Internet Protocol. 12, 112

IPC Inter-Process Communication. 18

ISA Instruction Set Architecture. 133, 155

ISO International Organization for Standardization. 138

JSON JavaScript Object Notation. 43

KB Kilobyte. 71

LAN Local Area Network. 41

LLVM Low Level Virtual Machine. 92, 93

LZMA Lempel-Ziv-Markov chain Algorithm. 63

MCU Microcontroller. 135, 137

MIPS Microprocessor without Interlocked Pipeline Stages. 63

MITM Machine-in-the-Middle. 57, 63–65

MMU Memory Management Unit. 15–17, 148

MPU Memory Protection Unit. 15, 17

NT Windows NT. 4

NTP Network Time Protocol. 2, 107

OLED Organic light-emitting diode. 148, 151, 152, 155

OS Operating System. 16, 19, 63, 78, 108, 109, 111, 113, 115, 116, 119

OTA Over-the-Air. 110

OTW One-Time Write. 110

PC Personal Computer. 85

PFS Perfect Forward Secrecy. 48

PROM Programmable Read-Only Memory. 19

RAM Random Access Memory. 13, 15, 16, 46, 71, 79, 136

RFC Request for Comments. 14, 15, 31, 55

RISC Reduced Instruction Set Computing. 133, 151

ROM Read-Only Memory. 19

RTOS Real-Time Operating System. 19

SDK Software Development Kit. 109

SoC System on a Chip. 134

SoK Systematization of Knowledge. 70

SPI Serial Peripheral Interface. 112, 136, 148, 149, 152, 155, 156

SRAM Static Random-Access Memory. 135

SSD Solid State Drive. 19

SSDP Simple Service Discovery Protocol. 54, 68, 69

SUIT Software Updates for Internet of Things. x, 23, 31–33, 71, 81, 82, 91

TCB Trusted Computing Base. 79, 112, 116

TCP Transmission Control Protocol. 12

TLS Transport Layer Security. x, 42–50, 56–61, 70–72, 102, 106, 107

TTL Transistor–Transistor Logic. 155

TUF The Update Framework. 24, 25, 32

TV Television. 51, 55, 56, 62, 65, 67

UART Universal Asynchronous Receiver / Transmitter. 112

UDP User Datagram Protocol. 49, 60

UK United Kingdom. 42

URL Uniform Resource Locator. 55, 62, 65, 67, 69

US United States. 42

USB Universal Serial Bus. 110

VM Virtual Machine. 35, 46

WAMR WebAssembly Micro Runtime. 115, 116, 138

WASI WebAssembly System Interface. 115

WASM WebAssembly. 133, 156

WAT WebAssembly Text format. 37

XML Extensible Markup Language. 43, 56, 62, 65