Escaping Vendor Mortality: A New Paradigm for Extending IoT Device Longevity

Conner Bradley
connerbradley@scs.carleton.ca
Carleton University
Ottawa, Ontario, Canada

David Barrera
davidbarrera@cunet.carleton.ca
Carleton University
Ottawa, Ontario, Canada

ABSTRACT

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 paper 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 paper helps guide future discussions on this important topic.

CCS CONCEPTS

• **Computer systems organization** → **Firmware**; • **Security and privacy** → *Operating systems security*; *Software security engineering*; • **Software and its engineering** → Open source model.

KEYWORDS

IoT, Software Updates, Longevity

ACM Reference Format:
Conner Bradley and David Barrera. 2023. Escaping Vendor Mortality: A New Paradigm for Extending IoT Device Longevity. In *New Security Paradigms Workshop (NSPW '23), September 18–21, 2023, Segovia, Spain.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3633500.3633501

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" [46], "75 billion by 2030" [58], "a whooping trillion connected devices by year 2035" [42]. 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 [33, 34]. 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) [18, 54]. 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 [28].

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 [2], 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 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 [36].

[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.

NSPW '23, September 18–21, 2023, Segovia, Spain

ACM ISBN 979-8-4007-1620-1/23/09.
https://doi.org/10.1145/3633500.3633501

In this paper, 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 [63].

This paper 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 trade-offs between centralized, distributed, and hybrid approaches for securing software updates.

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 [5, 69]. 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 [5, 69].

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).

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 [67], these devices are out of scope; Linux's heavy focus on preserving user space application binary interface (ABI) compatibility [61] 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 [9]. Challenges like storage capacity limitations, intermittent connectivity, and power constraints can hinder the firmware update process [32].

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 [50, 57, 62]. 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 [7, 40, 72]. 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

[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 [10].

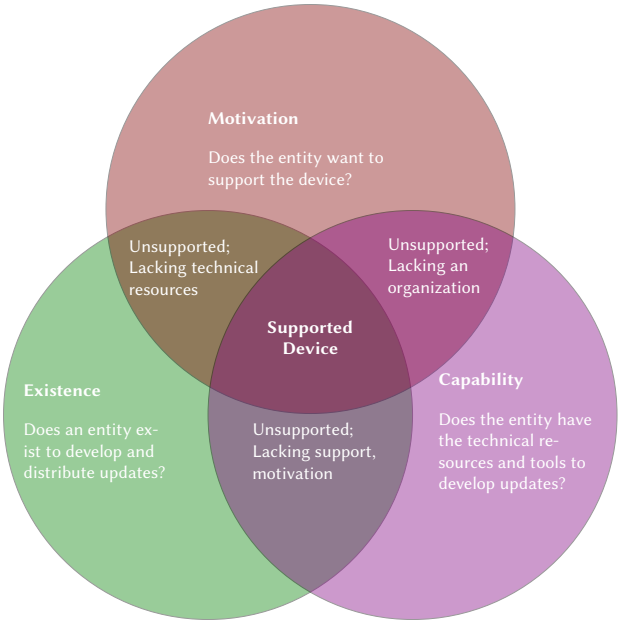

Figure 1: Whether a vendor can provide software updates depends on their existence, motivation, and ability. Without any one of these factors, a vendor's ability to support devices becomes hindered.

they rely upon [40]. Kiningham et al. [43] 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" [50] while implementing their update scheme. One worst-case estimate for the time horizon for quantum-accelerated key extraction is approximately 2030 [39]. Assuming this estimate holds true, 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.

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.

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 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 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, it is difficult to argue that this is not 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.

2.3 Software Updates within Walled Gardens

On personal computers (PC), 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 [37]. 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 NON-SOLUTIONS

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 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 3.1 and 3.2), establishing standardized mechanisms for secure firmware distribution (3.4), or delegating legacy maintenance to third parties (3.3 and 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.

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 [64]. 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 [35].

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 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.

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 resold 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.

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

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

[4]Centralized app stores can perform application vetting, including developer authentication, functionality checks, API tests, security tests, etc. [8]

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.

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 [62] 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 [50].

Unfortunately, these protocols deem the issue of device longevity out of scope [50], or ignore it altogether [62]. 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 [9, 50]. 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 [71].

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 [7, 40] 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.

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 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 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 5.

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 [25]. With this in mind, a common suggestion is for IoT devices to adopt a similar e-waste recycling strategy [64].

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 [28].

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

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 [28]. 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 [28]. 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 to their nearest recycling facility to responsibly dispose of the device[8]. 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 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 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* [21, 25], 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 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 [22]. 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 [27]. 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 [22, 27]; 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.

4.1 Obsolescence and the IoT Lifecycle

This section presents the IoT device lifecycle, as shown in Figure 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 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 [56]. 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.

[8]For cheap devices, fuel costs for transporting the device to a recycling facility may exceed the production cost of the device.

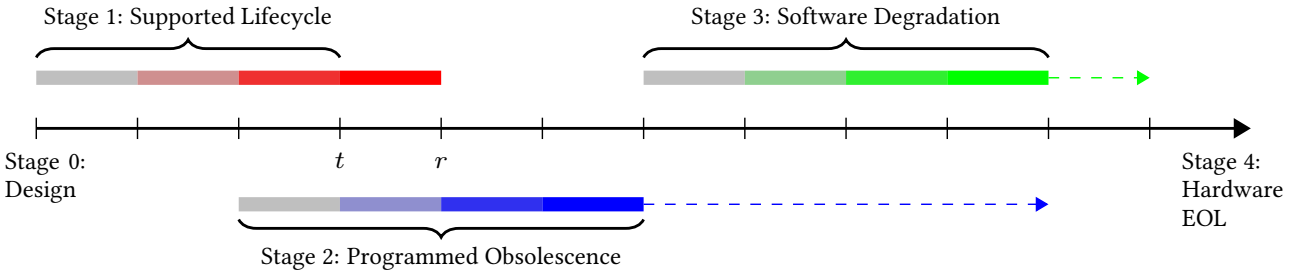

Figure 2: An IoT device's lifetime represented as an abstract timeline. From device inception, the first period t represents the end of software support from the first-party vendor, and r represents the point in time when a lack of software updates causes a degradation in device functionality. We propose adding a transition period where an IoT device can detect obsolescence, and then transition to a third-party development model.

Empirically, the velocity with which new devices are shipped appears to take priority over the design of a sound software update architecture [11].

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. [52] 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 [22]. 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 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 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 [38]. 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 [59]. 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 [24, 68]. 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 [51].

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 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 5.2. For example, the IoT device may not support the required cipher suites to connect to newer TLS endpoints [11], 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 [29]. 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* [22].

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 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 (see the challenges outlined in Section 3.6), 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. 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.

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.

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 hardware 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 [25].

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.

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

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[9] 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.

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 computers[10] 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. Existing IoT operating systems such as Contiki [26], RIOT [6], Tock [47], among many others tend to support a narrow range of microcontrollers and hardware, with no cross-compatibility between OSes [31]. 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 manufacturers 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 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[11], 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 [20], 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 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 (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 [20].

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 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 (OTA) updates, which, as previously mentioned in Section 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.

[9] Consider that not only is the root CA certificate required, but it also needs to be replaced when it expires.
[10] The early days of personal computers that included operating systems.
[11] The first-party vendor of the Android device itself, e.g., Samsung.

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.

6.1 Addressing the Maintenance Burden

Under our model, the first-party vendor is initially responsible for all layers of the device, as shown in Figure 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 (HAL), any proprietary firmware needed to enable hardware functionality, and some basic primitives for a microkernel-inspired OS [6, 44]. As shown in Figure 3, this would correspond to the middle portions of the operating system stack shown in blue. These layers can effectively contain all the first-party vendor's 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 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 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 bugs in the kernel. 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.

One shortcoming of this approach is the exclusive reliance on the first-party vendor's capability to provide updates to natively compiled code, represented by the blue components in Figure 3. Unfortunately, potential solutions to address this issue appear impractical and non-viable. For example, it seems unlikely that a vendor (even if compelled to) would release their kernel and other intellectual property publicly for other maintainers, as discussed in Section 3.3. To overcome this challenge, we suggest a compromise by implementing a microkernel architecture, reducing the amount of first-party code in the TCB. This minimizes complexity and the kernel's attack surface while abstracting functionality into platform-independent components. During the period of first-party vendor support, any bugs in the microkernel or proprietary firmware can

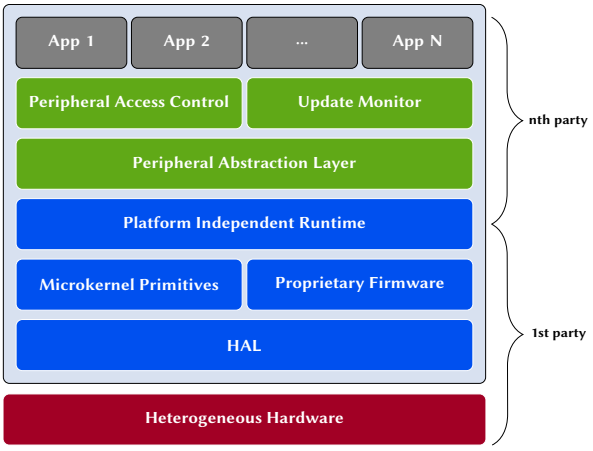

Figure 3: Our proposed development model splits responsibility between multiple parties. The creator of the device is responsible for creating the device's hardware and implementing a minimal OS and runtime. Platform-independent code can be created for multiple devices without proprietary tooling.

be addressed, but we cannot assume that all vendors will engage in providing consistent initial support for all IoT devices.

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 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 (TCB) 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 [44]; however, IoT deployment verticals are typically not high-performance, 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, Section 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 few lingering bugs would exist in the device after the first-party support period ends, and these bugs will not be high severity. 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, without relying on any platform-specific code. This design choice solves the issues highlighted in Section 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 [3] (WASI), which enables access to lower-level OS and hardware features. The emergence of the lightweight WebAssembly Micro Runtime (WAMR) [4], 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 [48, 53, 70], 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 [10] is crucial for its success.

One limitation of the platform-independent runtime as envisioned is its complexity. While our approach reduces 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 Sections 3.3 and 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.

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 6.3. 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.

Table 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.

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

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 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.

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 [41]). 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.

Note that not all third-party maintainers are open-source providers. When it comes to securing IoT devices, transitioning to an open-source third party may not align with the specific goals of the device deployment. Our model provides the flexibility to switch to a new third-party vendor, whether they offer company-based support or come from the open-source community, depending on the threat model and intended use case of the IoT device.

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 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 [43]. 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 paper, 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 [45] 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 [57]. Ultimately, a third-party centralized authority will allow for the better overall security of IoT devices[12]. This centralized model shares similarities with how UEFI-based systems distribute and install firmware updates, such as using the Linux Vendor Firmware Service for distribution and UEFI firmware capsules for installation [49, 65]. 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. However, decentralization on its own is not a panacea.

Transitioning from a first-party to a third-party source for model updates introduces the potential for new security vulnerabilities, such as those arising from open-source supply chains. Historical cases have shown malicious code being injected into open-source packages and contributions [17, 23], leading to significant security breaches. To address these risks, strict code vetting, authentication mechanisms, and continuous monitoring are essential to ensure the integrity and security of third-party contributions.

Trusted computing. Including trusted hardware like TrustZone or SGX can serve as a hardware root of trust for an IoT device. Depending on an IoT device's threat model, leveraging trusted computing technologies would be useful for isolating critical pieces of functionality to ensure confidentiality and integrity. However, trusted computing technologies have thus far been unable to demonstrate long-term resistance to vulnerabilities [14, 16, 30], making them unsuitable for our longevity goals. Without a hardware root of trust, devices may have to rely on software-only solutions, possibly with human confirmation of signature or key validation.

Additionally, first-party vendors could employ trusted computing technologies in a pervasive manner, which runs counter to the goal of our design. Trusted computing technology and other technology protection mechanisms have been employed to ensure vendor lock-in, which negatively affects consumers' right to repair. We discuss this issue further in Section 7.1.

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.

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

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.

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 [56, 63]. 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 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 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 [63]. 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 (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 [15, 19, 66]. 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 [15, 66].

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" [63]), 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.

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 [13]. 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 [13]. 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 [12, 55]. 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 [12]. 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 the company 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 [12, 55], 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.

7.3 Sustainable Design

Designing for sustainability can help IoT devices last longer, even after they become obsolete. Stead et al. [60] 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 [60] developed by Stead et al. could 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.

7.4 Motivating Vendor Adoption

Perhaps one of the largest drawbacks of our proposed design is the challenge of motivating vendor adoption. From a revenue perspective, vendors may be hesitant to spend time and resources on keeping legacy devices up to date. Moreover, if these legacy devices do not generate any direct revenue stream (e.g., subscription-based devices, see Section 3.1), there may be little motivation for vendors to implement a long-term support system. Instead, vendors prefer to deprecate old devices, forcing consumers to purchase new replacements that have all the latest features and security patches. The idea of extending the lifespan of devices may conflict with a vendor's profits, so it is unclear what motivation there would be for vendors to take this new paradigm seriously.

Vendor image and public perception are potential driving factors for this: consumers are increasingly concerned regarding the sustainability and longevity of IoT ecosystems, especially given that more and more devices are internet-connected and supposedly designed for long-term deployments [1]. For IoT devices to last long term, they should be usable long term. A vendor's overall image may be tarnished if they continually deprecate old products, which may not inspire confidence from future buyers [54].

To address the growing issue of e-waste from IoT devices [34], an idea has been proposed to implement e-waste recycling taxes on all electronic devices that require recycling [28]. This would increase product costs and generate funding for proper recycling, while also encouraging consumers to invest in longer-lasting products. If these recycling taxes increase substantially, more consumers will become more conscious about the longevity of the electronic devices that they depend on.

In summary, vendors are unlikely to adopt this design voluntarily. Rather, incentives in the form of policies need to be implemented to make it more cost-effective for them to comply or to provide other benefits for doing so.

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 paper 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. We've proposed an initial software stack for future devices as well as a set of technical mechanisms through which devices can securely switch to a new support channel once their current vendor becomes unavailable. We hope this paper serves as an initial step toward the goal of long-lasting IoT devices, and ultimately toward the sustainable use of the Internet of Things.

REFERENCES

[1] Jacob Abbott, Jayati Dev, DongInn Kim, Shakthidhar Reddy Gopavaram, Meera Iyer, Shivani Sadam, Shirang Mare, Tatiana Ringenberg, Vafa Andalibi, and L Jean Camp. 2023. Kids, Cats, and Control: Designing Privacy and Security Dashboards for IoT Home Devices. In *Proceedings 2023 Symposium on Usable Security*. Internet Society.

[2] Francisco Javier Acosta Padilla, Emmanuel Baccelli, Thomas Eichinger, and Kaspar Schleiser. 2016. The Future of IoT Software Must be Updated. (2016). https://hal.inria.fr/hal-01369681

[3] Bytecode Alliance. 2023. WebAssembly System Interface. https://github.com/WebAssembly/WASI

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

[5] Manos Antonakakis, Tim April, Michael Bailey, Matthew Bernhard, Elie Bursztein, Jaime Cochran, et al. 2017. Understanding the Mirai Botnet. (2017).

[6] Emmanuel Baccelli, Oliver Hahm, Mesut Günes, Matthias Wählisch, and Thomas C. Schmidt. 2013. RIOT OS: Towards an OS for the Internet of Things. In *2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 79–80. https://doi.org/10.1109/INFCOMW.2013.6970748

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

[8] David Barrera and Paul Van Oorschot. 2010. Secure software installation on smartphones. *IEEE Security & Privacy* 9, 3 (2010).

[9] Jan Bauwens, Peter Ruckebusch, Spilios Giannoulis, Ingrid Moerman, and Eli De Poorter. 2020. Over-the-Air Software Updates in the Internet of Things: An Overview of Key Principles. *IEEE Communications Magazine* 58, 2 (2020). https://doi.org/10.1109/MCOM.001.1900125

[10] Carsten Bormann, Mehmet Ersue, and Ari Keränen. 2014. Terminology for Constrained-Node Networks. RFC 7228. https://doi.org/10.17487/RFC7228

[11] Conner Bradley and David Barrera. 2023. Towards Characterizing IoT Software Update Practices. In *Foundations and Practice of Security*. Springer, 406–422. https://doi.org/10.1007/978-3-031-30122-3_25

[12] Plastic Break Free From. 2021. Brand Audit Report 2021. https://brandaudit.breakfreefromplastic.org/brand-audit-2021/

[13] Taylor Brydges. 2021. Closing the loop on take, make, waste: Investigating circular economy practices in the Swedish fashion industry. *Journal of Cleaner Production* 293 (2021). https://doi.org/10.1016/j.jclepro.2021.126245

[14] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. 2020. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1416–1432. https://doi.org/10.1109/SP40000.2020.00061

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

[16] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. 2019. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 142–157. https://doi.org/10.1109/EuroSP.2019.00020

[17] Monica Chin. 2021. How a university got itself banned from the linux kernel. https://www.theverge.com/2021/4/30/22410164/linux-kernel-university-of-minnesota-banned-open-source

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

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

[20] LineageOS Community. 2023. LineageOS Android Distribution. https://lineageos.org/

[21] Tim Cooper. 1994. Beyond recycling: The longer life option. (1994).

[22] Tim Cooper. 2004. Inadequate Life? Evidence of Consumer Attitudes to Product Obsolescence. *Journal of Consumer Policy* 27, 4 (2004). https://doi.org/10.1007/s10603-004-2284-6

[23] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the Impact of Security Vulnerabilities in the Npm Package Dependency Network. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18)*. Association for Computing Machinery, 181–191. https://doi.org/10.1145/3196398.3196401

[24] Daniel DiClerico. 2010. HP inkjet printer lawsuit reaches $5 million settlement. https://www.consumerreports.org/cro/news/2010/11/hp-inkjet-printer-lawsuit-reaches-5-million-settlement/index.htm

[25] Evelien Dils, John Bachér, Yoko Dams, Tom Duhoux, Yang Deng, and Tuli Teittinen. 2020. Electronics and obsolescence in a circular economy. https://www.eionet.europa.eu/etcs/etc-wmge/products/etc-wmge-reports/electronics-and-obsolescence-in-a-circular-economy

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

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

[28] Vanessa Forti, Cornelis Peter Baldé, Ruediger Kuehr, and Garam Bel. 2020. The global e-waste monitor 2020. *Quantities, flows, and the circular economy potential* (2020).

[29] Samuel Gibbs. 2014. Is the year 2038 problem the new Y2K Bug? https://www.theguardian.com/technology/2014/dec/17/is-the-year-2038-problem-the-new-y2k-bug

[30] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache Attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security (EuroSec'17)*. Association for Computing Machinery, Article 2. https://doi.org/10.1145/3065913.3065915

[31] Oliver Hahm, Emmanuel Baccelli, Hauke Petersen, and Nicolas Tsiftes. 2016. Operating Systems for Low-End Devices in the Internet of Things: A Survey. *IEEE Internet of Things Journal* 3, 5 (2016), 720–734. https://doi.org/10.1109/JIOT.2015.2505901

[32] José L. Hernández-Ramos, Gianmarco Baldini, Sara N. Matheu, and Antonio Skarmeta. 2020. Updating IoT devices: challenges and potential approaches. In *2020 Global Internet of Things Summit (GIoTS)*. IEEE. https://doi.org/10.1109/GIOTS49054.2020.9119514

[33] Stacey Higginbotham. 2018. The internet of trash [Internet of Everything]. *IEEE Spectrum* 55 (2018). https://doi.org/10.1109/MSPEC.2018.8362218

[34] Stacey Higginbotham. 2021. The IoT's E-Waste Problem Isn't Inevitable.

[35] Jeremy Hsu. 2018. Why the military can't quit windows XP. https://slate.com/technology/2018/06/why-the-military-cant-quit-windows-xp.html

[36] Muhammad Ibrahim, Andrea Continella, and Antonio Bianchi. 2023. AoT - Attack on Things: A security analysis of IoT firmware updates. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*.

[37] Apple Inc. [n. d.]. App Review - App Store. https://developer.apple.com/app-store/review/

[38] monitor io. 2023. End of service and instructions for a standalone option. https://www.monitor-io.com/

[39] Shuxian Jiang, Keith A. Britt, Alexander J. McCaskey, Travis S. Humble, and Sabre Kais. 2018. Quantum Annealing for Prime Factorization. *Scientific Reports* 8, 1 (2018). https://doi.org/10.1038/s41598-018-36058-z

[40] Falguni Jindal, Rishabh Jamar, and Prathamesh Churi. 2018. Future and challenges of internet of things. *Int. J. Comput. Sci. Inf. Technol* 10, 2 (2018), 13–25.

[41] Trishank Karthik, Akan Brown, Sebastien Awwad, Damon McCoy, Russ Bielawski, Cameron Mott, Sam Lauzon, André Weimerskirch, and Justin Cappos. 2016. Uptane: Securing software updates for automobiles. In *International Conference on Embedded Security in Cars*.

[42] Madhu Khurana, Thipendra Pal Singh, and Tanupriya Choudhury. 2021. Effective threat and security modelling approach to devise security rating of diverse IoT devices. In *Data Driven Approach Towards Disruptive Technologies: Proceedings of MIDAS 2020*. Springer, 583–593.

[43] Kevin Kiningham, Mark Horowitz, Philip Levis, and Dan Boneh. 2016. CESEL: Securing a Mote for 20 Years. In *Proceedings of the 2016 International Conference on Embedded Wireless Systems and Networks (EWSN '16)*. Junction Publishing.

[44] Gerwin Klein, Michael Norrish, Thomas Sewell, Harvey Tuch, Simon Winwood, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, and Rafal Kolanski. 2009. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles - SOSP '09*. Association for Computing Machinery, 207. https://doi.org/10.1145/1629575.1629596

[45] Antonio Langiu, Carlo Alberto Boano, Markus Schuß, and Kay Römer. 2019. UpKit: An Open-Source, Portable, and Lightweight Update Framework for Constrained IoT Devices. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. https://doi.org/10.1109/ICDCS.2019.00207

[46] Oded Leiba, Yechiav Yitzchak, Ron Bitton, Asaf Nadler, and Asaf Shabtai. 2018. 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)*. IEEE, 29–39.

[47] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*. Association for Computing Machinery. https://doi.org/10.1145/3132747.3132786

[48] Renju Liu, Luis Garcia, and Mani Srivastava. 2021. Aerogel: Lightweight Access Control Framework for WebAssembly-Based Bare-Metal IoT Devices. In *2021 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE. https://doi.org/10.1145/3453142.3491282

[49] LVFS 2023. Linux Vendor Firmware Service. https://fwupd.org/

[50] Brendan Moran, Hannes Tschofenig, David Brown, and Milosch Meriac. 2021. *A Firmware Update Architecture for Internet of Things.* Request for Comments RFC 9019. Internet Engineering Task Force. https://doi.org/10.17487/RFC9019

[51] Pierre-Emmanuel Moyse. 2020. The Uneasy Case of Programmed Obsolescence. *UNBLJ* 71 (2020), 61.

[52] Shinsuke Murakami, Masahiro Oguchi, Tomohiro Tasaki, Ichiro Daigo, and Seiji Hashimoto. 2010. Lifespan of Commodities, Part I. *Journal of Industrial Ecology* 14, 4 (2010). https://doi.org/10.1111/j.1530-9290.2010.00250.x

[53] Niko Mäkitalo, Tommi Mikkonen, Cesare Pautasso, Victor Bankowski, Paulius Daubaris, Risto Mikkola, and Oleg Beletski. 2021. WebAssembly Modules as Lightweight Containers for Liquid IoT Applications. In *Web Engineering*. Springer.

[54] Jay Peters. 2020. Google discontinues its Google Nest Secure Alarm System. https://www.theverge.com/2020/10/19/21523967/google-discontinues-nest-secure-alarm-system

[55] Marta Pizzetti, Lucia Gatti, and Peter Seele. 2019. Firms Talk, Suppliers Walk: Analyzing the Locus of Greenwashing in the Blame Game and Introducing 'Vicarious Greenwashing'. *Journal of Business Ethics* 170, 1 (2019). https://doi.org/10.1007/s10551-019-04406-2

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

[57] Justin Samuel, Nick Mathewson, Justin Cappos, and Roger Dingledine. 2010. Survivable key compromise in software update systems. In *ACM conference on Computer and Communications Security*. Association for Computing Machinery. https://doi.org/10.1145/1866307.1866315

[58] Mustafizur R Shahid, Gregory Blanc, Zonghua Zhang, and Hervé Debar. 2018. IoT devices recognition through network traffic analysis. In *2018 IEEE international conference on big data (big data)*. IEEE, 5187–5192.

[59] Amazon staff. 2023. Our decision to wind down Amazon Halo. https://www.aboutamazon.com/news/company-news/amazon-halo-discontinued

[60] Michael Robert Stead, Paul Coulton, Joseph Galen Lindley, and Claire Coulton. 2019. The little book of sustainability for the Internet of Things.

[61] Linus Torvalds. 2012. Linux Kernel Mailing List: Media commit causes user space to misbahave. https://lkml.org/lkml/2012/12/23/75

[62] David Tracey and Cormac Sreenan. 2017. OMA LWM2M in a holistic architecture for the Internet of Things. In *2017 IEEE 14th International Conference on Networking, Sensing and Control (ICNSC)*. IEEE. https://doi.org/10.1109/icnsc.2017.8000091

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

[64] Hannes Tschofenig and Stephen Farrell. 2017. Report from the Internet of Things Software Update (IoTSU) Workshop 2016. RFC 8240. https://doi.org/10.17487/RFC8240

[65] UEFI Forum, Inc. 2022. *Unified Extensible Firmware Interface (UEFI) Specification.*

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

[67] Sebastian Vasile, David Oswald, and Tom Chothia. 2019. 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. https://doi.org/10.1007/978-3-030-15462-2_12

[68] Steven Vaughan-Nichols. 2021. No ink, no scan: Canon USA printers hit with class-action suit. https://www.zdnet.com/article/untrustworthy-canon-printer-lawsuit/

[69] Aohui Wang, Ruigang Liang, Xiaokang Liu, Yingjun Zhang, Kai Chen, and Jin Li. 2017. An Inside Look at IoT Malware. In *Industrial IoT Technologies and Applications (LNICST)*. Springer. https://doi.org/10.1007/978-3-319-60753-5_19

[70] Elliott Wen and Gerald Weber. 2020. Wasmachine: Bring iot up to speed with a webassembly os. In *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE.

[71] Glenn Wurster and P. C. van Oorschot. 2008. The Developer is the Enemy. In *Proceedings of the 2008 New Security Paradigms Workshop (NSPW '08)*. Association for Computing Machinery, 89–97. https://doi.org/10.1145/1595676.1595691

[72] Peter Zdankin and Torben Weis. 2020. Longevity of Smart Homes. In *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. https://doi.org/10.1109/PerComWorkshops48775.2020.9156155