Capture: Centralized Library Management for Heterogeneous IoT Devices

Han Zhang
Carnegie Mellon University

Abhijith Anilkumar
Carnegie Mellon University

Matt Fredrikson
Carnegie Mellon University

Yuvraj Agarwal
Carnegie Mellon University

Abstract

With their growing popularity, Internet-of-Things (IoT) devices have become attractive targets for attack. Like most modern software systems, IoT device firmware depends on external third-party libraries extensively, increasing the attack surface of IoT devices. Furthermore, we find that the risk is compounded by inconsistent library management practices and delays in applying security updates—sometimes hundreds of days behind the public availability of critical patches—by device vendors. Worse yet, because these dependencies are "baked into" the vendor-controlled firmware, even security-conscious users are unable to take matters into their own hands when it comes to good security hygiene.

We present *Capture*, a novel architecture for deploying IoT device firmware that addresses this problem by allowing devices on a local network to leverage a *centralized* hub with third-party libraries that are managed and kept up-to-date by a single trusted entity. An IoT device supporting Capture comprises of two components: Capture-enabled firmware on the device and a remote driver that uses third-party libraries on the Capture hub in the local network. To ensure isolation, we introduce a novel *Virtual Device Entity* (VDE) interface that facilitates access control between mutually-distrustful devices that reside on the same hub. Our evaluation on a prototype implementation of Capture, along with 9 devices and 3 automation applets ported to our framework, shows that our approach incurs low overhead in most cases (<15% increased latency, <10% additional resources). We show that a single Capture Hub with modest hardware can support hundreds of devices, keeping their shared libraries up-to-date.

1 Introduction

With their growing popularity, in-home Internet-of-Things (IoT) devices are becoming ripe victims for remote attacks, leading to high-profile incidents such as the Mirai botnet [5]. Compared to traditional network hosts, IoT devices are often more vulnerable due to weak credentials [40, 60, 83], insecure protocols [43], and outdated software [57, 61]. Making matters worse, despite their deployment in homes, these devices may connect directly to public Internet hosts to send data and even listen for incoming connections [30, 64]. If any of them are compromised, attackers can easily wreak further havoc by moving on to other devices on the same network [5, 80].

Although many current IoT exploits originate from misconfigurations, weak credentials, and insecure applications [3, 40], the extensive use of third-party libraries in IoT devices may have security implications but remains overlooked. Vulnerabilities in common libraries, when left unpatched, can affect a massive number of devices (e.g., CallStrager [82] and Ripple20 [84]). The security impact of vulnerable libraries in *traditional* software systems is well-known [12, 14], with slow rollout of security-critical patches exacerbating the issue [21, 41, 49]. To understand whether this situation is as common in IoT, we conducted a study of 122 IoT firmware (Section 3), finding widespread use of common libraries. Matching firmware release dates to CVE disclosures, we observed significant delays in patching critical vulnerabilities (up to 1454 days), and inconsistent patch rollout even across the same vendor. As end-users are usually unable to address these vulnerabilities themselves, our findings call for better ways of managing third-party IoT libraries, mitigating potential threats arising from vulnerable libraries in the future.

Recent works in IoT security may partially alleviate this challenge, but each has its limitations (Table 1). Commercial IoT frameworks and operating systems (e.g., Microsoft Azure Sphere [48], AWS Greengrass [4], and Particle Device OS [53]) all assume the burden of managing a *limited* set of shared libraries provided by the OS. However, developers may use a variety of IoT libraries for functionality [54]. These OSes provide little protection for those custom libraries imported by developers. Alternatively, several proposals attempt to isolate vulnerable devices on the network [22, 36, 70]. Network isolation offers limited flexibility when it comes to mitigating the effects of compromised devices, so these approaches present an inherent security tradeoff whenever devices need Internet access.

	Automated Library Updates	Prevent Malicious Network Access	Secure Custom Libraries	No Firmware Changes	No Application Code Changes
Commercial IoT OS [4,48,53]	✓	✗	✗	✗	✗
Network Isolation [22,36,70]	✗	✓	Partial	✓	✓
Capture	✓	✓	✓	✗	Optional

Table 1: Comparing Capture with other IoT security approaches. Commercial IoT OSes offer centralized management for a limited set of libraries. Network isolation blocks unnecessary network communications, limiting exposure of vulnerable libraries. Unless developed natively, existing IoT devices need to modify firmware to include either commercial OSes or Capture runtime. Application code built with existing OS APIs also needs to change accordingly for the new OS's APIs; for Capture, some integration approaches provide backward-compatible API interfaces, avoiding changes to app-level code.

We present *Capture*, an approach that aims to reduce the IoT attack surface stemming from vulnerable third-party libraries without compromising functionality. Capture is a novel software architecture for writing IoT firmware, which enables centralized management of third-party libraries, thus simplifying the deployment of security-critical patches to home IoT devices. Rather than a monolithic firmware running on an IoT device, Capture partitions firmware across the device and its driver on a central hub. The hub is a trusted entity under users' direct control and maintains libraries updated. When developing Capture-enabled devices, vendors can implement the remote driver to use libraries maintained by the hub rather than managing updates individually for each version of their firmware. To provide flexibility and backwards-compatibility, Capture still allows developers to deploy custom, "unsupported" libraries directly on the device firmware, but leverages isolation to reduce the attack surface and limit damages to others in case they become compromised.

To realize this vision, we must address several challenges. First, since Capture splits devices into local firmware and drivers, traditional device network identifiers such as MAC and IP addresses are too coarse. Instead, we propose a novel abstraction, *Virtual Device Entity*s (VDE) — a combination of device, driver, and associated accounts and network configurations on the hub — as the basis for managing devices across hardware and facilitating access control.

Second, since we move part of the firmware functionality from device hardware onto a shared, centralized hub, we must ensure that drivers running on the hub are properly isolated from each other such that they function the same way as they did on the dedicated hardware. This is especially important so that even if a device is compromised it cannot affect the other devices on the hub. We place every VDE into its own subnet attached to a unique virtual network interface (vNIC). By blocking inter-vNIC traffic, we prevent devices from sending network packets to each other. We also assign unique user accounts and utilize Linux security primitives to isolate shared resources on the hub.

Finally, as Capture represents a significant shift in the conventional IoT architecture for developers, we take steps to simplify the migration of existing IoT devices to our framework. We design and evaluate three integration approaches based on how current IoT devices are implemented — OS Default Library Replacement, Existing IoT Framework Extension, and Native Driver Development — showing that Capture can be adopted by developers by changing a few lines of code in their existing firmware.

We developed a prototype Capture Hub on a Raspberry Pi 3 (RPi 3), and migrated 9 open-source IoT applications ranging from streaming cameras to extensible "smart" mirror displays into the framework. These applications cover a variety of hardware platforms, from embedded real-time micro-controllers to fully-provisioned Linux installations. In addition, we implemented 3 home automation applets on IFTTT [38], which provide additional macro-benchmarking data. Our evaluation shows that porting an application is often straightforward, while using Capture introduces a modest latency increase (15% on average, $<$23 ms in most cases). We believe this is imperceptible from a typical user's perspective, although it may vary depending on the set of applications that are installed. In particular, for IoT automation platforms such as IFTTT, the overhead of Capture is negligible compared to the time needed to communicate with the cloud backend. Applications that rely on throughput also fare well, experiencing 34% overhead on average, which we found preserves qualitative functionality. Importantly, our results show that the hub itself scales well to many devices: the inexpensive RPi 3 prototype can easily accommodate on the order of 50 devices without over-subscription, with more capable hardware allowing hundreds of independent devices.

In summary, we make the following contributions:

- We present Capture, a novel architecture for deploying IoT firmware in a way that supports centralized management of third-party libraries, thus eliminating the need for timely updates from individual vendors.
- We introduce Virtual Device Entities (VDEs) to securely manage devices in Capture, and isolate untrusted components running on shared hardware from each other.
- We propose three integration approaches for migrating existing IoT devices to Capture. Our evaluation on 9 open-source IoT devices shows that these apps can be

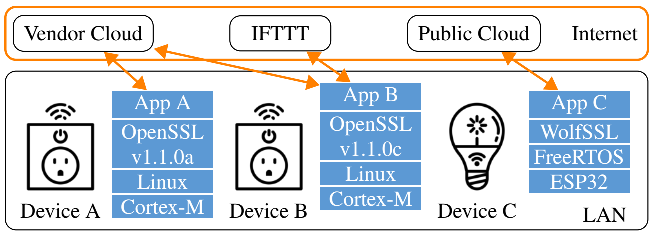


Figure 1: Current IoT device software stacks and network communication. Devices have a variety of platforms (ARM Cortex-M, ESP32) but utilize similar third-party libraries.

migrated to Capture with minimal changes.

- We implement a prototype of Capture on a RPi 3 and evaluate its performance for 9 IoT apps and 3 IFTTT applets. We show that Capture incurs low performance overhead (<15% latency increases and <10% extra on-device resources on average) and a single Capture Hub can support dozens to hundreds of local devices. The code is available at `https://github.com/synergylabs/iot-capture`.

2 Background and Setting

In this section, we provide essential background on the IoT setting that we assume for the rest of the paper. Interested readers are encouraged to read the comprehensive SoK paper by Alrawi et al. [3] for additional details on IoT deployments and security considerations.

IoT Device Software Stack. Figure 1 illustrates three representative IoT devices and their software architecture based on teardown blogs [1, 18]. IoT devices use a variety of microcontrollers (MCUs) with different capabilities. For example, devices using ARM Cortex-M MCU can run a version of Linux, supporting numerous Linux libraries (e.g., OpenSSL). Meanwhile, more inexpensive devices often use less capable MCUs, such as Espressif ESP-32 with 520 KB RAM [26]. They also use light-weighted RTOSes and libraries (e.g., wolfSSL) to reduce resource use. Given that IoT developers often focus their effort on building compelling application software (e.g. App A, B, C in the figure), alternative IoT platform designs have been proposed (e.g., HomeOS [19], Azure Sphere [48], Particle OS [53]) which offer low-level OS and library security updates as a service, enabling developers to focus on applications using a limited set of APIs.

Home IoT Networking. During the installation of a device in their home, users typically connect IoT devices to the Internet either directly by associating them with their home WiFi router, or through a vendor-provided hub (e.g. Samsung's SmartThings hub or the Philips Hue bridge) which is then cloud-connected. Internet-connected devices can be publicly accessible (via Network Address Translation (NAT) from routers) due to functionality requirements, but may be reachable from Internet attackers as well [11, 81]. Although sometimes devices can be restricted to not access the Internet, they can still communicate with other devices on the LAN without users' involvement using, for example, the UPnP protocol [40, 42]. This can lead to cross-device exploits and escalation attacks [5, 82].

Figure 1 shows an example IoT home deployment with three devices that communicate with external hosts, including the vendor's proprietary cloud, the IFTTT automation service, and possibly generic cloud service providers such as AWS or Azure. In this example, however, not all devices are equally secure. Device A and Device B both use OpenSSL, but Device A uses an outdated version (1.1.0a) as compared to Device B (1.1.0c). Device C, which runs on limited hardware, makes use of a lighter-weight SSL library (WolfSSL). Even in a small deployment, it may be common to see a wide range of security-critical third-party libraries in use, becoming even more of an issue in realistic settings.

3 Third-Party Libraries in IoT

In this section, we seek to address two key questions which are largely unanswered. Namely, *how prevalent is third-party library usage among existing IoT devices*, and *how diligent are device vendors when it comes to releasing firmware updates that patch critical security vulnerabilities?*

Previous studies [10, 49, 86] that focus mainly on network equipment report widespread vulnerabilities, some of which can be attributed to unpatched third-party libraries. A recent study focusing on smart appliances reports similar findings [3]. However, these studies do not address the state of affairs on current IoT devices, and in particular on how frequently libraries are used and updated. To fill this gap in our knowledge, we conducted a measurement study on 122 firmware releases from 26 devices and 5 popular vendors. We find that third-party library use is prevalent, and even more concerning, that security-essential libraries like OpenSSL often remain unpatched for hundreds of days.

3.1 Data Collection

Retrieving Library Information. A potential approach is to analyze the binary images of publicly available firmware images. However, despite the availability of analysis tools [20, 33], validating the resulting information would be time-consuming and error-prone, and the number of devices with easily obtainable firmware images is limited. Instead, we collect vendor-reported information about the use of GPL open-source libraries in firmware release notes, as this disclosure is required by the license terms. While our results may thus exclude information about closed-source and non-GPL

Vendor	BLK	TP	Ring	Nest	D-Link	Total
Devices	12	3	1	7	3	26
Firmware	12	3	1	74	32	122
Libraries	80	5	53	290	93	441
Lib. versions	103	5	55	400	114	654

Table 2: Summary of devices and vendors included in the measurement. We skip firmware for network equipment since our focus is on smart devices. BLK — Belkin, TP — TP-Link.

third-party libraries, we note that this will, if anything, under-represent the true prevalence of third-party library use in IoT devices. We used this approach to collect all available data for 441 unique libraries across 122 firmware releases from 5 IoT vendors, dating back to 2011. We manually collected library names and version numbers for 122 firmware releases.

Firmware Selection. We selected 5 popular device vendors (Belkin, TP-Link, Ring, Nest, and D-Link) since we were able to find consistent, detailed information about their firmware releases with the required third-party library information. Table 2 summarizes 122 firmware releases we collected data about. Nest and D-Link provide the most comprehensive information about their firmware release history, dating back to 2011. We use these historical releases to analyze longitudinal patching behaviors. Belkin and TP-Link maintain public information for a single firmware version for each device still under support. Ring releases one summary for all open-source libraries used in their devices, which we categorize as a single generic device with a single firmware release. Table 3 includes individual device details.

3.2 Results

From the collected data, we aim to characterize two main statistics: the prevalence of third-party library usage in IoT firmware images across vendors, and the characteristics of patch release over time. In particular, our goal for the latter statistic is to understand how quickly a new firmware image is released after a third-party library is updated in response to a known CVE with a corresponding moderate or high severity.

Prevalence. As expected, we found that IoT devices use third-party libraries extensively. Table 2 shows that the 122 firmware releases we studied disclosed 441 unique open-source libraries. Counting libraries with different version numbers as unique, this number increases to 654. While some vendors consistently use the same version across images, others do not: for example, of the 12 Belkin devices we studied (each corresponding to one image), there are 80 unique libraries spanning 103 library versions. This finding already suggests problematic patching behavior. While there is a significant variation in the range of libraries in use (441 across just 26 devices), there is a common subset across devices. Figure 2 shows the most popular libraries, appearing in at least 50% of the devices. OpenSSL and BusyBox are ubiquitous, used by 92.31% and 88.46% out of a total of 26 devices.

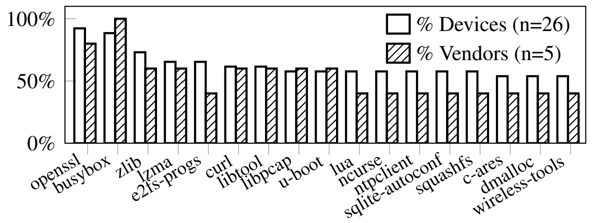


Figure 2: List of the most common libraries in all 26 devices across vendors. Among 26 devices, over 50% use these libraries. The most popular ones, OpenSSL and BusyBox, are used by 92.31% and 88.46% of devices. We also show the percentage of vendors who use these libraries on their devices.

Patching Practices. To better understand the security risk of third-party library use, we examine firmware releases longitudinally, and their alignment with library patches and CVE disclosures. Since historical release data was only available for 5 devices from Nest and D-Link, we use 100 firmware releases for these devices, for a 7-year period (2011-2018).

We pick OpenSSL to study library patching practices for two reasons. First, OpenSSL is a popular library used by all vendors in our dataset, except for Ring which uses GnuTLS. Second, OpenSSL is critical for software security and has a well-documented history of vulnerability discoveries and patches [52]. By examining OpenSSL versions in firmware releases and OpenSSL's update history, we analyze vendors' patching behaviors and outstanding vulnerabilities over time.

Figure 3 shows the "age" of the OpenSSL library, defined as the number of days elapsed since the release date of a particular version. The dashed lines represent the library ages used in different device firmware, while the solid green lines represent the ideal case where the devices can always use the *most up-to-date* library versions. As shown in these dashed lines, device firmware updates routinely lag behind using the latest versions of OpenSSL. In some cases, this extends for hundreds of days. For example, Nest Protect's last firmware release on 2016-07-13 used a 1525 days old OpenSSL version, while the latest available one was released on 2016-05-03 (only 71 days old). Furthermore, there are often multiple new firmware releases made by vendors without incorporating the up-to-date library version, suggesting a missed opportunity. Notably, even devices from the *same* vendor often use different library versions, highlighting the challenge of coordinating upgrades.

The Nest Learning Thermostat appears to have the best patching practices among devices in our study; it sometimes even used the latest OpenSSL (red circles in Figure 3). However, a closer look at how this aligns with known vulnerabili-

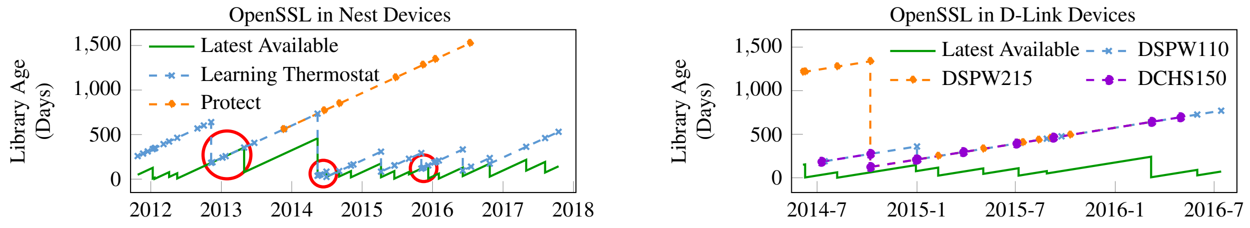


Figure 3: OpenSSL library ages in different devices. Dashed lines represent actual library used in the firmware. Each marker indicates a new firmware release. Solid lines indicate the expected library age **if** new firmware release always uses the latest versions, representing a best-case scenario. Red circles highlight cases in which devices *actually* use the latest version.

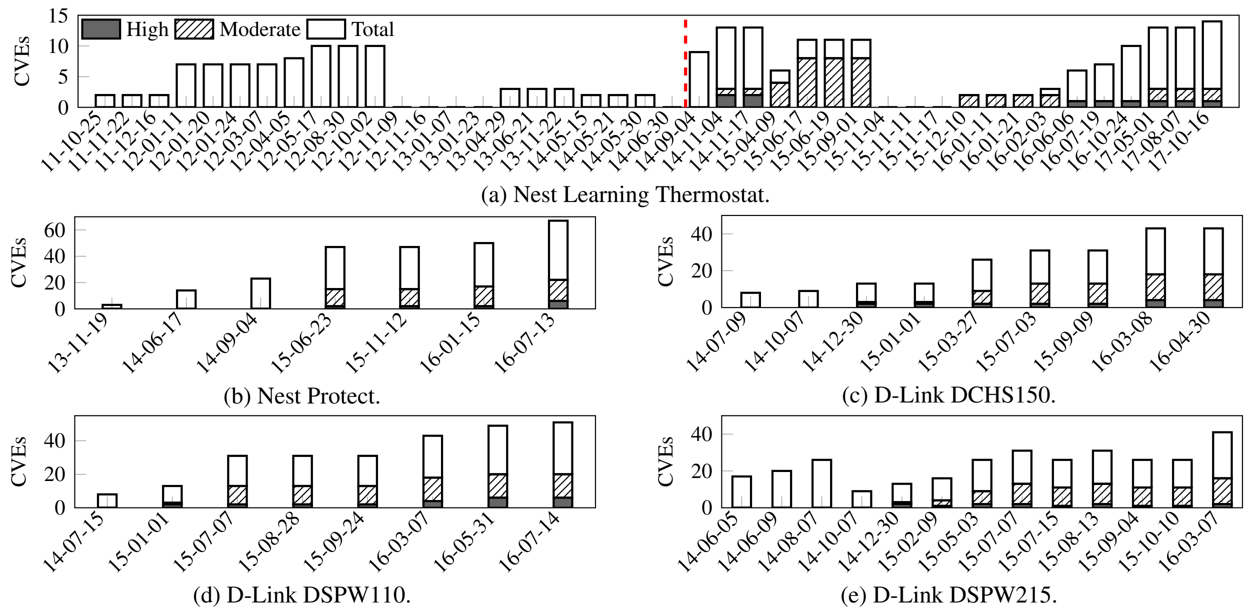


Figure 4: Number of publicly known OpenSSL CVEs in firmware releases. X-axis shows the firmware release date. We do not have CVE severity breakdowns for data prior to August 2014 (the red dashed line in (a)). For newer libraries, we find many High and Moderate CVEs present in the firmware. Certain Nest Protect firmware releases are skipped due to missing release dates.

ties suggests that even this case reflects unnecessary exposure. Figure 4a depicts the number of OpenSSL CVEs and in particular those of moderate or high severity (severity data is only available after August 2014), that apply to each version of the Nest Learning Thermostat in this time frame. Unsurprisingly, the periods corresponding to Figure 3's red circles are not vulnerable, but this only lasts for a few months until multiple vulnerabilities emerge. Importantly, most of these CVEs are avoidable only if the firmware uses the latest OpenSSL.

Hardware Architecture. Many devices in our dataset are Unix-based systems, as 88.46% and 46.15% of devices include BusyBox and Linux Kernel libraries. Teardowns on high-end smart devices [17, 18, 35] often find powerful ARM processors, affirming our findings. Meanwhile, budget-oriented devices may prefer alternative microcontrollers (such as ESP32 and ESP8266 in light bulbs and plugs [1, 2]). Our dataset might under-represent lower-end devices for two reasons. First, they could use libraries provided by chip maker, royalty-free [27]. Second, we had some difficulty searching for open source compliance notices from several lesser-known vendors.

Key Takeaways and Limitations. Our measurement results reveal concerning statistics about the current state of third-party library management in IoT devices. Just by considering widely used open-source GPL libraries, we show that even market-leading vendors such as Nest and D-Link oftentimes fail to update their dependent libraries promptly. This results in unnecessary exposure to known vulnerabilities. While our data collection methodology is limited to open-source GPL libraries, we aim to shed light on the existing state of IoT library mismanagement using these libraries as indicators.

Device	Vendor	Firmware	Release Date	Libraries	Library Versions
WeMo F7C027/F7C028	Belkin	1	2019/08/09	53	55
Wemo Light Switch v1 F7C030	Belkin	1	2019/08/09	53	55
WeMo SNS	Belkin	1	2015/10/14	53	54
WeMo Mini F7C063	Belkin	1	2019/09/05	54	54
WeMo Smart	Belkin	1	2015/06/30	54	55
WeMo Smart F7C046/47/49/50	Belkin	1	2019/09/05	53	54
WeMo WLS040	Belkin	1	2019/09/04	55	55
WeMo Dimmer	Belkin	1	2019/09/03	47	48
WeMo InsightCR	Belkin	1	2019/08/09	53	54
WeMo Jarden	Belkin	1	2019/09/03	53	54
WeMo Maker	Belkin	1	2019/09/03	53	54
WeMo Insight F7C029	Belkin	1	2019/08/09	53	54
SmartPlug - HS100	TP-Link	1	N/A	5	5
SmartPlug - HS110	TP-Link	1	N/A	5	5
SmartPlug - HS200	TP-Link	1	N/A	5	5
Generic Release	Ring	1	N/A	53	55
Nest Cam	Nest	2	N/A	177	186
Nest Connect	Nest	1	N/A	7	8
Nest Detect	Nest	1	N/A	12	13
Nest Guard	Nest	1	N/A	107	108
Nest Hello	Nest	1	N/A	20	20
Nest Learning Thermostat	Nest	57	2011/10/25 - 2017/10/16	140	194
Nest Protect	Nest	11	2013/11/19 - 2016/07/13	18	21
DSPW110	D-Link	9	2014/07/15 - 2016/07/14	75	86
DSPW215	D-Link	14	2014/06/05 - 2016/03/07	72	85
DCHS150	D-Link	9	2014/07/09 - 2016/04/30	51	54

Table 3: Details of devices and firmware releases included in the measurement. For each device, we count the number of unique libraries and unique library-version combinations across all firmware releases.

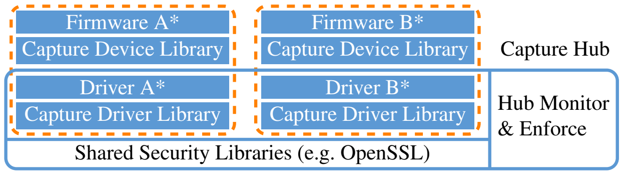


Figure 5: Capture system architecture. Every device consists local device firmware and driver on the hub. They form a logical unified entity, Virtual Device Entity (orange dashed box). The Capture Hub maintains a central version of common libraries and has extra monitoring and enforce modules.

4 Capture Framework

To mitigate the security threats from outdated libraries in device firmware reported in Section 3, we present Capture, a novel architecture for deploying IoT firmware to support centralized management of third-party libraries, alleviating the need for library updates by individual vendors.

4.1 Overview

Figure 5 provides an overview of Capture. A Capture Hub in the local network centralizes library security updates. Every device has two components: a device firmware (F/W A*, B*), and a remote driver (Driver A*, B*) running on the Capture Hub. Developers can use default drivers (provided by Capture) or implement custom ones to use the latest libraries on the hub. The device firmware and the driver use Capture SDK libraries for network communication. Moreover, the driver uses API wrappers provided by Capture to interact with common libraries on the hub. If vendors need libraries not provided by Capture, they can include custom dependencies in their firmware while still benefiting from Capture's isolation protection. The Capture Hub Monitor and Enforce module manages all drivers and provides runtime and network isolation for all devices supported by it.

Threat Model. We assume that the Capture Hub is trusted, and all standard wireless protocols and Linux tools we use to provide isolation are up-to-date to address any vulnerabilities. We consider an adversary who seeks to compromise IoT devices through known vulnerabilities in unpatched third-party libraries. Unlike prior efforts that restrict devices to explicitly whitelisted hosts (e.g., the vendors' cloud backend) [36, 39], we allow devices to communicate with arbitrary hosts to avoid limiting their functionality. Since local devices (or drivers) may be compromised, our goal is to prevent them from being able to affect other non-compromised devices and drivers in the same home deployment. Attack vectors from zero-day exploits (i.e. no patches available) and non-library vulnerabilities (e.g., weak passwords) are out of the scope of this work. In addition, we exclude side-channel attacks arising from the shared hub access from different drivers.

Security Goals. Intuitively, the main goal of Capture is to **centralize library management** by providing a consistent, up-to-date set of third-party libraries for devices in the local network, configured and managed by the central hub. Since we do this by splitting the firmware across an IoT device and a hub, Capture should not introduce *new* vulnerabilities or attack opportunities. For example, Capture needs to preserve **device integrity** by protecting communication that would normally be on the device. Hence Capture needs to prevent any entity from intercepting or impersonating a device with its driver on the hub and vice-versa.

In addition, Capture needs to maintain proper **isolation** between devices and drivers. This implies that compromised *devices* should not be able to communicate with other hosts on the same local network, and that compromised *drivers* on the hub should not affect the operation of devices other than the one that they represent.

4.2 Library Update Management

Capture alleviates the burden of patching security-critical shared libraries, enabling device vendors to use the up-to-date versions on the Capture Hub without managing patches themselves. Notably, vendors still implement their device firmware and the corresponding drivers. However, they may be concerned with losing control over devices' stability whenever Capture automatically updates shared libraries. These library updates can potentially cause semantic changes (e.g., new APIs) or unexpected bugs to break the existing functionality of the drivers. Fortunately, prior work on patching vulnerable libraries for Android apps provides an optimistic outlook [16], reporting that 97.8% of apps using libraries with known vulnerabilities can be fixed with a drop-in patched version of the library. To determine whether this finding applies to IoT devices, we analyze the dataset from Section 3 for potential impacts of library updates on device functionality. We focus on the OpenSSL library usage in Nest devices, since their dataset has a comprehensive history of versions and upgrades.

OpenSSL Versioning. OpenSSL's versioning scheme uses letters to denote minor security patches and numbers for major changes [51]. For example, an application using version 1.0.2a can upgrade to 1.0.2b to fix bugs and security vulnerabilities, while an upgrade to 1.1.0 indicates new features and APIs. Each major version has an end-of-life date, after which users stop receiving security updates. OpenSSL's staggered release strategy supports multiple major versions at the same time, providing a buffer to transition between versions. Our analysis on Nest's OpenSSL use finds that Nest always upgrades the major version before the old one reaches end-of-life.

Library Update Strategies. There are three strategies for Capture to support multiple library versions concurrently.

Maintain Multiple Majors in Parallel. The most stable strategy to preserve device functionality is to support all active major versions in parallel. The hub applies security patches for each major version independently. According to the OpenSSL's release history [52,76], Capture has to support two or three majors concurrently and needs to apply security updates every few months. This strategy will not break any Nest device's functionality in our dataset, since they never use any outdated major versions.

Only Maintain the Latest Major Version. Managing multiple library versions in parallel may become complicated as the number of libraries increase. A simple strategy is to only keep one version per library on the hub, presumably the latest major release. Based on our dataset, Nest devices use a non-latest major version in 1238 out of 2184 days. This strategy will cause version mismatches almost half of the time. Mixing drivers intended for older versions with newer runtime can be problematic. Although OpenSSL meticulously preserves backward compatibility across major upgrades [51], we are pessimistic about third-party libraries ' stability in general. Therefore, we use the major mismatch as a *conservative estimator* of potential functionality breakages. Choosing how many major versions to support demonstrates the tradeoffs between manageability and functionality.

Forceful Major Upgrades after End-of-Life. Vendors could ignore library upgrades so long that it reaches the end-of-life dates. Capture could forcefully upgrade major versions to maintain security at the expense of potential functionality breakages. Since Nest always upgrades OpenSSL to the next major version before the end-of-life dates, we do not have data to measure the impact of a forceful upgrade. However, this tradeoff is a very difficult yet open challenge. Prior works proposed various strategies from blocking devices with insecure libraries [39], quarantining insecure devices locally [22], to preserving functionality at the expense of security [43]. We plan to leave this as a configurable option for end-users to make informed decisions based on their concerns.

4.3 Virtual Device Entities (VDEs)

An IoT device supporting Capture comprises of two components: a Capture-enabled firmware on the device and an associated software driver running on a hub, collectively forming a *Virtual Device Entity* (VDE). Note that Capture creates a unique VDE instance for every deployed device. Even if there are multiple identical devices, Capture instantiates separate VDE instances for each of them. Capture ensures confidentiality within the VDE and enforces isolation across different VDEs, as we will explain in the following sections.

Device Bootstrap. Figure 6 illustrates the process of bootstrapping new devices and obtaining VDE. A device first connects to a *setup network* with pre-shared credentials, just

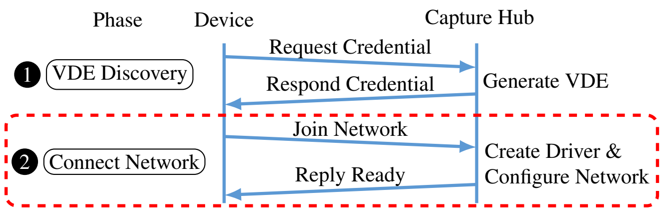


Figure 6: Device bootstrap procedure. In Step 1, the device connects to the Capture Hub using a shared setup network. Then it joins a VDE-specific VLAN network in Step 2 (dashed box). Section 4.4 discusses more details on network configurations. Section 4.6 addresses potential attacks during bootstrap.

like traditional home WiFi. In Step ①, the Capture Hub creates a fresh VDE and prepares a VDE-specific VLAN on the second operation network. After receiving the VDE-specific credential, the device disconnects the setup network and joins the *operation network* (Step ②), where the hub binds the device to its VLAN. This transition won't affect other existing devices, since they are connected to their VDE-specific VLANs already. The hub creates a driver for the VDE, sets up network interfaces and isolation, and enforces resource isolation for the driver on the hub.

4.4 Communication Isolation

A Capture-enabled device essentially functions as a "local" device since it can only communicate with its driver on Capture Hub and vice versa. Other communication, such as between local devices or different drivers, is automatically blocked. We achieve this in Capture by creating unique logical networks for each VDE with its own subnet and virtual interface.

The Capture Hub simultaneously manages two separate WiFi Access Points (APs). The first one is a WPA2-Personal AP with pre-shared credentials for the first step of initialization (Figure 6), similar to current home WiFi. The second AP uses WPA2-Enterprise and enforces VDE-based isolation. Specifically, Capture Hub creates unique RADIUS user accounts and constructs different virtual Network Interface Cards (vNICs) for each VDE. Using enterprise features such as VLAN and RADIUS authentication, the second AP binds each VDE's device into its own subnet and vNIC. The hub binds the corresponding driver to the same vNIC interface using TOMOYO [68], a Linux security module for mandatory access control. If the driver needs Internet access, the hub creates a designated public-facing port and enables the driver's connection to the port via TOMOYO. We then configure the firewall program `iptables`'s rulesets to block communications across vNICs to achieve VDE-based isolation. Capture's VDE-based isolation is inspired by DreamCatcher [22], which shows vNIC-based isolation is effective against link-layer spoofing. We extend DreamCatcher's network isolation with additional mandatory access control to accommodate Capture Hub's shared driver execution environment.

To bind multiple devices into different vNICs while using a single WiFi AP, we utilize the VLAN isolation feature from WPA2-Enterprise. While WPA2-Personal is common for home users, popular WiFi modules used by vendors to build their products already support WPA2-Enterprise [24]. Hence we believe modern devices can support Capture and WPA2-Enterprise either out of the box or with updated firmware. For legacy devices without WPA2-Enterprise support, Capture can create a new WPA2-Personal AP for each legacy device, however that may run into software limitations of the number of SSIDs per antenna [22]. An alternative approach is to create unique WPA group keys for each device, isolating hosts under one shared WPA2-Personal network [70]. Capture didn't take this approach as it requires modifying standard protocols.

4.5 Resource Isolation

Since Capture Hub executes multiple drivers, a key challenge is to ensure secure and fair resource sharing on the hub. Capture needs to ensure slow or malicious drivers are contained and cannot affect other VDE's availability and private data. Linux containers [45] seem like a natural choice for process isolation. However, they are ill-suited for Capture since each container has a copy of the libraries the driver needs. Whenever the library is updated, all container images would have to be updated and rebuilt, which conflicts with our goal of managing libraries centrally. Instead, Capture provides resource isolation and access control using lightweight Linux system primitives. The Capture Hub creates a new Linux user account per VDE, under which context the associated driver runs, applying standard Linux filesystem and memory protections. We further limit the driver's capability by utilizing the TOMOYO Linux extension and its domain-based security management. We assign each VDE and all of its subprocesses to the same security domain and enforce security policies for networking and file systems. Finally, we used Linux `cgroups` [34], a key building block for implementing containers, to limit the resources used by each VDE. Linux cgroups are known to be an efficient and low overhead mechanism to account for resource usage [55, 79]. Currently we statically set the CPU and the memory resources for each driver to equally share the total system resources, but in the future, we can add support for drivers to specify their resource demands (such as via manifest files during installation, similar to mobile apps) and dynamically enforce them.

4.6 Security Analysis

External Threats. Capture protects devices from external threats by securing the driver components, which are reachable from the Internet. This is done by the Capture Hub, which ensures that the latest library versions are installed automatically and used by the drivers, without the device vendors having to do this. Unlike drivers, the actual devices are isolated from other hosts in the local network. Manufacturers still implement custom firmware running on their devices, meaning that some outdated libraries and vulnerabilities may still exist. However, since the network isolation in Capture only allows communication between driver and device, it limits other hosts from exploiting them. This security protection is contingent on vendor adoptions and properly implemented driver software.

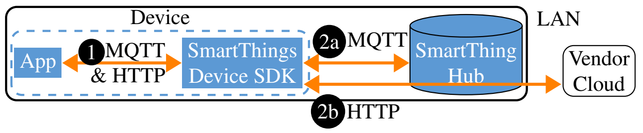
Internal Threats. We consider internal threats which include compromised devices, drivers, and other devices within the WiFi range. Capture prevents compromised local devices from attacking other Virtual Device Entities (VDEs) through network isolation since these devices are confined to their VDE and cannot reach any other hosts directly. Similarly, a compromised driver is also isolated from other VDE drivers using our network and other resource isolation mechanisms (Mandatory Access Control, cgroups) mentioned above. In Capture, drivers communicate with their associated device using our library runtime, which requires developers to specify the message format between the device firmware and driver. This design prohibits compromised drivers from sensing arbitrary packets to their associated devices and affecting them. Furthermore, drivers cannot communicate with other VDEs on the hub due to our resource isolation mechanisms.

Malicious devices (including Capture-incompatible local devices) can not learn about other VDE's network credentials simply by eavesdropping on the setup network. Although the setup network is a WPA2-Personal AP with shared password credentials, each device actually has its own PTK (pairwise transient key) through WPA2 4-way handshake [43, 75]. However, link-layer encryption provided by WPA2 is insufficient for Capture's network isolation because all drivers will run in the same application layer on the hub. Therefore, we generate a unique network interface and VLAN for each VDE during the bootstrap process (Figure 6).

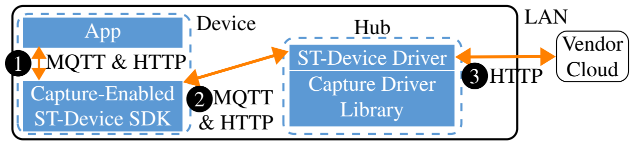
An adversary could potentially impersonate the Capture Hub and perform man-in-the-middle attacks during new device bootstraps (Figure 6). This threat can be mitigated by using certificates and public key infrastructure for devices to verify the hub's identity, or other novel device pairing and initialization techniques [32, 65]. We did not implement these features in our prototype since our current threat model focuses on attacks from vulnerable third-party libraries (Section 4.1).

5 Integration Approaches

We propose three integration approaches for developers to adopt Capture, motivated by current IoT development practices. Our goal is to provide paths of least resistance to help



(a) Current deployment requires SmartThings Hub for networking.



(b) Capture-enabled SmartThings devices move all network communication onto the device drivers at the central hub.

Figure 7: Integration using IoT framework SDK extension.

with the adoption while providing flexibility to developers.

5.1 OS Library Replacement

The first approach is to provide a Capture-enabled version of standard OS libraries. Take the OS networking library in ESP32 platform, `WiFi.h`, for example. Devices use APIs from this library to connect access points, maintain web servers, and communicate over sockets. We provided a fully-compatible Capture-enabled library, named as `CaptureWiFi.h`. Developers just need to make minor changes to use Capture, such as replacing the `#include <WiFi.h>` statement and initializing Capture global runtime. We provide a default Capture driver on the hub, which acts as a proxy to relay network traffic. If the original device works as a webserver, we open a public-facing server on the driver to forward traffic and restrict network traffic between driver and device.

This approach is platform-dependent. We need custom implementations for specific OS APIs and libraries. However, this is a one-time effort that can then be used by device developers with minimal porting effort. For example, all of our prototype apps use the same ESP32 modified library runtime.

5.2 IoT Framework SDK Extension

Similar to replacing OS APIs, our second approach is to extend the SDK of a popular IoT framework to support Capture. IoT frameworks (e.g., Azure Sphere [48], Particle OS [53], and Samsung SmartThings Device SDK (ST-SDK) [62]) provide rich functionalities to differentiate from standalone embedded device OSes with limited networking APIs. For example, Azure Sphere [48] and Particle DeviceOS [53] provide APIs to communicate with their native cloud backends; Samsung SmartThings Device SDK [62] offers local devices the option of using the SmartThings Hub as an MQTT broker.

In this case, the developers of the IoT frameworks can incorporate Capture by modifying their SDK implementation while preserving existing functionality. As a proof of concept, we added Capture support into the ST-SDK, which enables third-party devices to use their SmartThings Hub. Figure 7a shows how an example device would integrate with the ST-SDK, similar to a custom OS library. A locally installed SmartThings Hub (ST-Hub) provides functions such as MQTT brokers, which device developers can directly invoke using ST-SDK APIs. A device-side library manages the underlying connections with the ST-Hub. We develop a Capture-augmented ST-SDK library (Figure 7b), so that device developers only need to switch their ST-SDK library runtime without modifying their application. Since the SmartThings Hub is proprietary, we were only able to re-create their known functions such as MQTT brokers using corresponding open source versions. We provide a default SmartThings-compatible driver to mimic the ST-Hub operations in Capture.

5.3 Native Driver Development

The two prior approaches provide default drivers on the Capture Hub to aid developer adoption. As a complementary approach, we developed a Capture Native Driver SDK, for developers to implement their own custom drivers with much more flexibility. To motivate this, consider an IoT device with a web server. Using the previous approaches, the default driver on our hub will create another public accessible web server for new connections, and relay incoming client connections to the device local-only web server. However, this may cause unnecessary latency to serve the web request since both inbound and outbound traffic has to go through the hub and processed by two webservers. To address this, we propose the Capture Native Driver SDK for developers looking to build customized drivers. Developers can use our SDK APIs to access security and networking functions on the Capture Hub, and even offload some computation to the hub.

6 Implementation

6.1 Core Hub Functionality

We implement the Capture Hub using a Raspberry Pi 3B+ with Linux in 1874 lines of C++ (`https://github.com/synergylabs/iot-capture`). We use the TOMOYO Linux security module [68] and `iptables` to implement the Virtual Device Entity based isolation mechanisms. Our hub uses hostapd [37] to manage WPA2 Personal and Enterprise WiFi APs. The main Capture program listens for new connections on the setup network, and upon request, creates a new VDE for the incoming device, allocates a new VLAN subnet with fresh RADIUS credentials, launches the corresponding driver program based on the device type, and updates the TOMOYO and `iptables` rulesets accordingly. The main program stores all metadata for each VDE locally. While our current prototype does not address device removal, this functionality can be added in a straightforward manner.

Optimizations. Existing applications often use blocking network calls. During prototype development, we observed a pathological case wherein the application only communicated using one sequential byte at a time. Clearly, adapting such applications into Capture introduces a significant performance penalty, as each read request will incur one round of communication with the driver residing on the hub. We found that without correction, this can lead to a 9.56x latency penalty for the simple Web Server app (listed in Section 6.2).

The first optimization we perform to address this issue is to introduce read and write buffers on the device. When an Internet host sends data to the driver, the payload is forwarded to the local device in batch. Subsequent read calls from the device will just retrieve the payload from the local buffer. Similarly, using write buffers enables network writes to be non-blocking I/Os, aggregating multiple payloads into chunks in one round of driver communication. We found that this reduces the latency penalty for the Web Server app from 9.56x to 1.62x, largely due to the reduced number of round trips to the hub.

Although the previous approach reduces average latency overhead to an acceptable 1.62x, it still incurs a median increase of 31 ms. We were able to attribute this to the poor wireless performance on the budget-oriented ESP32 microcontroller, where a single packet transmission can take up to 6 milliseconds. To reduce the total number of packets sent, we extended the protocol header fields and aggressively coalesce small packets throughout our protocol. One concrete example is proactively loading read buffers after accepting new clients, where previously the device needs to send two messages to check client status and fetch data to read, respectively.

Applying protocol optimizations and message coalescing bring down the median latency overhead to 1.2x (+10 ms), using the Web Server's baseline performance as a reference. Given that the ESP32 takes 5-6 ms to send a single packet, this approaches the limit of what can be done without better hardware. Detailed results are discussed in Section 7.1.

6.2 Benchmark Applications

To evaluate Capture, and explore different approaches for integrating apps, we developed 9 prototype applications (Table 4), including smart devices, Linux applications, and IoT frameworks, and 3 IFTTT automation applets for benchmarking (Table 5). Capture provides runtime libraries for device firmware and drivers to handle network setup and communication with the hub. The device-side library was implemented in 1335 lines of C++ code while the driver-side library varies.

Prototype Apps. We collected 6 open source applications from popular online forums and tutorials [23, 31, 71], and adapted them to use Capture. We chose the Espressif ESP32 platform given its reported popularity [1, 2] and use in hundreds of millions of IoT devices [25]. We implemented a

Abbreviation	App Name	Platform	Description
WEB	Web Server	ESP32	Standard web server to display and manage GPIO on/off status.
CAM	Camera	ESP32	Stream live video, take pictures.
SM	Servo Motor	ESP32	Adjust the speed of a servo motor.
CP	Color Picker	ESP32	Change the color of LED light bulb.
WS	Weather Station	ESP23	Monitor weather with a BME sensor.
TH	Temperature & Humidity	ESP32	Display temperature and humidity data from DHT sensor.
ST-L	SmartThings Lamp	SmartThings	Subscribe to MQTT broker to receive on/off message.
ST-S	SmartThings Switch	SmartThings	Publish to MQTT broker to issue on/off message.
MM	MagicMirror	Linux	Smart mirror display with online data such as news and weather.

Table 4: Prototype applications and descriptions.

generic default driver to support the OS Replacement approach, which required 166 lines of Python.

IoT Framework. We extended the Samsung SmartThings Device SDK (ST-SDK) [62] to showcase integrating Capture with existing frameworks (Section 5.2). ST-SDK is open-source, whereas other proprietary alternatives (e.g., Azure Sphere and Particle OS) raise challenges for replication and comparison. Capture-enabled devices cannot work directly with unmodified SmartThing Hub, so we analyzed ST-SDK's codebase and replicated its functionality with a driver that executes on the Capture hub. We then adapted sample applications provided by ST-SDK [63] into Capture.

Linux apps. Some IoT devices are powerful enough to run a Linux OS and applications (Section 3), so we adapted Linux smart devices into Capture to demonstrate its capability. We selected MagicMirror, a project with over 12K Github stars [47], that uses Raspberry Pi with a display to function as a smart mirror, displaying custom content (e.g. news and weather). Internally, the app includes a webserver and a browser to display the webpage. We migrated MagicMirror into a Capture prototype using the custom driver integration (Section 5.3) and separated the server component to the driver on the hub, keeping the display parts on the firmware.

Automation Applets. To better measure Capture's macro-benchmark performance impact on real-world scenarios, we implemented several home automation applets developed for IFTTT [38]. Prior work [46] categorized IFTTT applets by trigger-action service types (Device⇒WebApp, WebApp⇒Device, Device⇒Device) and reported an average execution latency of several seconds. We implemented Capture-enabled devices for all three trigger-action service types (Table 5), using the Web Server app (c.f. Table 4, WEB) on ESP32 in place of physical lights and switches, since it can control GPIO pins. Since ESP32 boards are lower performance and slow at performing SSL encryption, integrating these devices into Capture often improves performance due to our hub hardware being more capable. To provide a fair comparison, we also implement "mock" lights and switches directly on the Raspberry Pi and measure the latency impact from Capture integration as well.

7 Evaluation

Our evaluation aims to answer three primary questions.

- How much performance overhead do key device functionalities incur on Capture versus their native platform, and is the amount tolerable for typical home use?
- Can the Capture Hub scale to home deployments with hundreds of devices in the near future, and how many devices can our prototype reliably support at once?
- Roughly how much effort is required to port existing IoT devices to Capture, and do the integration approaches in Section 5 entail meaningful differences in the effort?

Our experiments were performed in a laboratory setting on 9 prototype devices (Table 4) and 3 IFTTT automation applets (Table 5). We use one Raspberry Pi 3 B+ as the Capture Hub and another Raspberry Pi and multiple ESP32 boards for prototype apps. Our evaluation results show that Capture typically incurs low overhead (15% latency increase, 10% device resource utilization), insignificant impact on applets from real-world automation platforms, and can support hundreds of devices for a single Capture Hub.

7.1 Performance Overhead

Setup. We compare the performance of apps running on Capture to that achieved by their original implementations. Because many IoT devices and automation apps are event-driven, they usually transmit a small amount of traffic but are sensitive to delays in latency. We categorize prototype apps (Table 4) into two categories: latency-sensitive and throughput-sensitive. We measure application-layer latency for all of them, but only measure the throughput reduction for the second group (such as a streaming camera). For most applications, we use Apache JMeter [6] to benchmark *average* and *median* latency for 500 HTTP requests. For the streaming camera (CAM), we measure the video latency by pointing the camera towards a millisecond clock and calculate average delays from 50 readings. For the SmartThings apps (ST-L and ST-S), we add instrumentation to send a notification packet to the hub so that we can calculate the time duration between the first MQTT message and the final notification from Wireshark's packet capture history. Finally, we measure the firmware code size and memory utilization on the device.

Simple Integration with All Apps. We aim to conservatively estimate Capture's performance impact assuming minimal burden on the developer. Hence, we first try to integrate apps with either OS or SDK replacements, since these require minimal modifications by the developer. If this attempt fails (for example, the app requires features not supported by our current prototype), we develop simple native drivers without spending too much engineering effort on app-specific optimizations.

Figure 8a shows the normalized latency for integrated apps. On average, apps experience a 15% latency increase due to the extra processing by the drivers on the hub. The baseline apps for the comparison process everything on the device and communicate directly with external hosts. After Capture integration, external hosts need the drivers on the hub acting as a proxy. For example, the camera streaming app driver needs to retrieve the raw footage from the device and forward it to the viewers. These extra steps introduce overhead to the end application. However, as Figure 8a shows, most apps experience a modest latency change between -34 ms and $+23$ ms. Given most apps' event-driven nature, this minor increase in absolute latency should not impact the quality of services for end applications. CAM app experiences the most substantial latency increase, increasing from 523 ms to an average of 820 ms (+297 ms), and a 40% FPS throughput reduction. However, the relative increase (1.6x) is on par with other apps. Since the baseline latency is very high, we believe the original app is not designed to be real-time for ESP32, and thus we did not further optimize its driver.

Several of the apps integrated with OS-Replacement see improved *average* latency results. This is because Capture-integrated apps perform more consistently, while the ESP32-only baselines occasionally experience latency spikes (thus having higher average results). *Median* results are more robust against outliers, and confirm Capture often increases latency slightly. The overall results show that Capture offers comparable performance to the baseline for most requests.

We measure the throughput overhead for several throughput-sensitive apps and report results in Figure 8b. For throughput metrics, we choose FPS for streaming, packet transfer rates for taking pictures, and full web page load time for the complex MagicMirror dashboard. The Camera app has a modest throughput reduction of around 40%. We observe no throughout drop for the Linux-based MagicMirror benchmark. Figure 8c shows that the Capture firmware is, on average, 10% larger and uses 7% more *on device* memory. We only measure the code increase for ESP-based devices given they have limited flash storage.

7.2 Overhead Perceived in the Real World

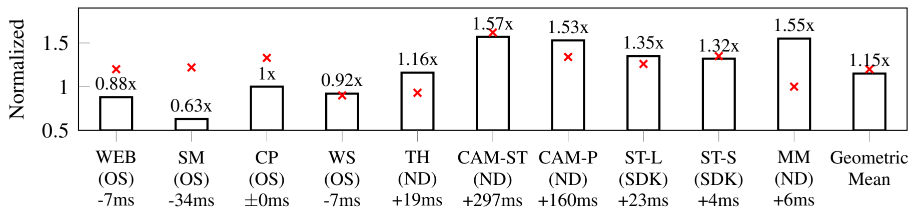
We implemented several IFTTT automation applets and measured Capture's impact on latency (Table 5). We programmatically trigger applets 30 times, reporting the average end-to-end latency. These results show moderate variances, largely due to the fact that these applets interact with remote cloud services (IFTTT, Google Sheets, and email servers), which is consistent with results from prior work [46]. Applets A1 and A2 show insignificant latency changes from Capture integration, indicating the communications to Internet services as the performance bottleneck. Applet A3's ESP32 integration demonstrates a benefit of Capture for low-budget devices. A3-ESP32 baseline has high latency due to compute-intensive tasks such as TLS encryption, while A3-Raspberry Pi and Capture-integrated ones have comparable latency results.

7.3 Scalability

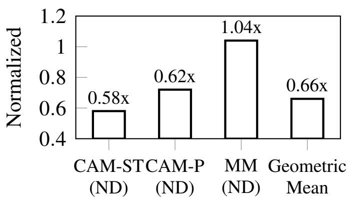
Since our Capture Hub executes all drivers on the hub, its resources limit the number of devices it can support. Among resources including memory, CPU, network interfaces, and private IP addresses, we identify the memory capacity as the key scaling bottleneck. The default driver for OS replacement uses the least amount of memory (3.7 MB) while the MagicMirror's driver uses the most memory (42 MB) as reported by `smem`'s Proportional Set Size [69]. Therefore, we emulated a deployment of 40 devices using the default drivers and 10 devices with MagicMirror drivers on a single Raspberry Pi 3B unit (1 GB RAM, quad-core). This setup uses 664 MB memory, but the CPU load average never exceeds 0.8 (max 4.0, due to four cores). Network virtual interfaces and subnets do not impose any practical limits with fine-grained assignments [58]. While the RAM on the hub is a limiting factor, several inexpensive platforms exist with more memory (e.g., Raspberry Pi 4 with 8 GB RAM for $75 [56]), which can potentially support hundreds of devices.

7.4 Integration Efforts and Tradeoffs

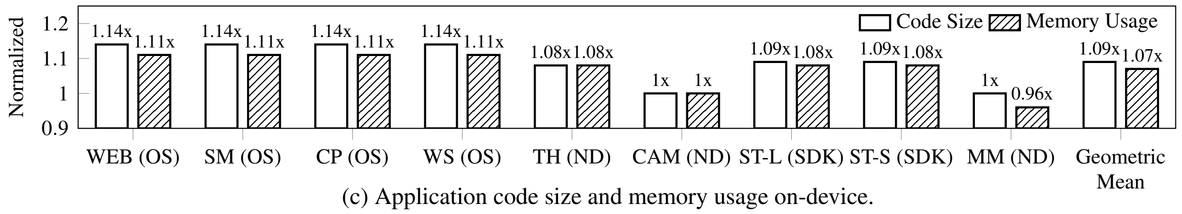
Integrating apps by replacing OS libraries or framework SDKs is straightforward, requiring modifying less than 10 lines of code after importing the Capture device library. Developing native drivers is more involved since it requires declaring a custom message format for device-driver communications and implementing the driver while delegating the network management to Capture's library runtime. The most sophisticated CAM driver we implemented was 817 lines of Python.



(a) Normalized average latency and numerical differences. Red crosses show *median* latency changes.



(b) Normalized average throughput.



(c) Application code size and memory usage on-device.

Figure 8: Performance overhead for all prototype apps. Data are normalized to results from the orignal apps. CAM has two modes: STreaming videos and taking Pictures. We denote integration approaches in parentheses: OS Replacement, Native Driver, and Framework SDK Replacement. Based on geometric means, Figure (a) shows a 15% latency increase and Figure (b) shows a 34% throughput reduction. Figure (c) shows the Capture-enabled firmware incur around 10% more on-device resource utilization.

ID	Service Type		IFTTT Applet Rule	ESP32 (seconds)		Raspberry Pi (seconds)	
	Trigger	Action		Original	Capture	Original	Capture
A1	Device	Web App	Turn on switch. ⇒ Add line to Google Sheet.	2.65±0.42	2.00±0.35	2.04±0.66	1.83±0.75
A2	Web App	Device	New email arrives inbox. ⇒ Turn on light bulb.	2.93±0.82	2.93±0.90	2.62±0.62	2.83±0.87
A3	Device	Device	Turn on switch. ⇒ Turn on light bulb.	2.21±0.43	0.81±0.16	0.94±0.28	0.88±0.35

Table 5: Average latency for automation apps with standard deviations (30 runs). Overall, Capture has insignificant impacts, with noteworthy improvements on A1 and A3 (ESP) due to offloading TLS operations on the hub. See Section 7.2 for further analysis.

We demonstrate the tradeoff between ease of adoption and performance impact by analyzing different integration approaches for the Web Server app. Although we spent considerable effort optimizing the default OS-replacement driver, it yielded a modest 12% average latency reduction over the baseline ESP32 app. The integration only requires changing a few lines of the original code. In comparison, implementing a native driver for this app significantly reduces latency by 36% over the same baseline. However, to implement the driver, we modified 264 lines of source code to process device-driver communication and customize protocols.

8 Limitations and Future Work

Vendor Incentives and Adoption Challenges. Vendors may be incentivized to use Capture because they can offload the security upkeep responsibility to a central trusted entity (the Capture Hub). They no longer need to keep applying security patches themselves, a task they often lag behind (Section 3). Capture's isolation design also helps protect vendors from other compromised devices in the user's local home.

There might be several hurdles for vendor's adoption. We have already proposed various integration approaches Section 5 to reduce adoption costs for existing devices and hub's library management strategies Section 4.2 to alleviate vendor's loss of agency and to avoid breaking functions.

The need for firmware splitting may pose another major roadblock for vendors. They have to bear the extra onus of developing two separated pieces of the "device" and the additional overhead in signing and logistics involved in firmware updates. Implementing Capture drivers and new firmware would require vendors to change significantly from the current status quo and would induce extra engineering efforts.

Single Point of Failure. Capture's centralized design means that the Capture Hub is a potential single point of failure; this is part of our threat model (Section 4.1), where the hub is assumed to be trustworthy. If the hub is compromised by vulnerabilities or privilege escalation bugs like those on conventional systems [9, 13], the integrity and confidentiality of the installed devices will be likewise compromised. By centralizing the management of security-critical updates, and providing additional isolation between devices, we hope to contribute to improving the overall security posture of devices deployed within the network (i.e., relative to the status quo). However, this improvement is contingent on vendor adoption.

Centralization may lead to a less robust network even without adversarial compromise. If the hub goes down, devices would lose network connectivity and drivers become unresponsive. Because most device firmware controls local actions (e.g., managing the on/off states for smart plugs), most devices should still function (e.g., through physical buttons on the device). Capture Hub failures, in this case, largely resemble network outages and router failures in current smart homes.

Protocol Compatibility. Since Capture isolates devices, link layer discovery and local network scanning no longer work. One such example is UPnP, an infamous protocol for posing security threats in IoT devices [40, 42] and recent exploits like CallStranger [82]. A future direction for our work is to provide a secure centralized discovery service on the Capture Hub itself with co-located drivers and shared libraries, substituting link layer discovery and mitigating fallout like CallStranger. With that said, many smart devices have companion smartphone apps that communicate with the device via a cloud service to support access to the device behind a home NAT. As communication through the cloud will not be impeded by our approach, we believe that the practical impact of Capture's isolation on everyday use will be minimal.

There are other potential security improvements, which are out of the scope of the current security goals for Capture and threat model. We do not support alternative wireless protocols such as BLE, Zigbee, and Z-Wave since Internet-based attacks over WiFi, the focus of our work, impose significant threats already. As future work, we can look into incorporating related works in securing other wireless protocols [36, 85] into Capture's centralized hub design. In addition, Capture does not address potential attacks due to weak security practices, such as the use of default credentials. However, Capture's Virtual Device Entity isolation blocks compromised devices from exploiting any other devices' vulnerabilities.

Augmenting Device Resources. Another opportunity that we have not explored is to use the hub's computation resources to augment the limited resources of local devices. Specifically, by introducing additional Capture APIs, we can extend the storage and processing capability of low-power microprocessors on the device to the hub.

Firmware Splitting. Capture proposes splitting monolithic firmware into remote and local components, an approach that could face practical challenges, such as data serialization, consistency, and fault tolerance. These issues are not uncommon to many distributed systems that make use of RPC-like components and have been studied extensively [7, 28, 29, 59, 67, 72–74, 77]. While our prototype implementation does not make use of all of these advances, Capture can benefit from this work to enhance its robustness and reliability. We view this as important future work.

9 Related Work

IoT Network Security. Several prior efforts have looked at IoT security issues [80], and proposed augmenting current network designs to address them. Dreamcatcher [22] uses a network attribution method to prevent link-layer spoofing attacks. Simpson et al. [66], DeadBolt [39], and SecWIR [43] propose adding features and components on network routers to secure unencrypted traffic. HoMonit [85], Bark [36], and HanGuard [15] propose finer-grained network filtering rules and context-rich firewall designs.

Capture takes a similar network-based approach drawing inspiration from isolation techniques used in prior works [8, 22, 70]. However, we take a more direct and principled approach to reduce the attack surface by centralizing standard library management. Centralizing shared libraries introduces additional challenges, which previous work does not consider.

IoT Software Security. Several projects address vulnerabilities in various aspects of current IoT software development. Vigilia [70] introduces capability-based network access control to protect devices while supporting home automation applications. Each device has one *driver* program, which provides public APIs accessible by home automation programs. In comparison, Capture focuses on security issues in traditional smart device firmware; by decoupling networking components in the original firmware into their drivers, Capture provides a centralized mechanism for updating shared libraries across all devices. Other efforts [44, 50, 78] address security challenges in the application-layer of devices, such as operation logging, cloud backend services, and automation apps, which are complementary to our work.

IoT Frameworks and OSes. Both academia and industry have looked at the challenges of IoT software stacks for smart homes with heterogeneous IoT devices. HomeOS [19] proposes a unified PC-like platform to manage all local devices. Commercial IoT frameworks emphasize their security offerings and ease of management for third-party developers. Microsoft Azure Sphere [48], Particle OS [53], and AWS Greengrass [4] all provide services to manage device library updates on behalf of developers. These frameworks also include native support for application-level over-the-air upgrades, reducing the barrier for developers to patch bugs. Samsung SmartThings Device SDK [62] reduces the developer burden of managing library updates by directly offering high-level APIs in the SDK (e.g., MQTT services). Developers do not need to worry about patching libraries, as long as they regularly update the SDK runtime.

While these frameworks help alleviate some of the developers' burden of library management, Capture offers several additional benefits. First, Capture has a secure isolation mechanism to protect against local malicious devices. Existing frameworks cannot offer isolation since they manage devices from public cloud backends. Second, Capture devices can install custom libraries on devices' firmware based on their requirements. Even if these libraries are vulnerable, attackers cannot exploit these libraries due to the isolation we provide. Third, as an open system, Capture's integration approaches are cross-platform and do not require device vendors to lock in to specific embedded system OSes and chipsets. Finally, IoT frameworks (Particle Device OS, Azure Sphere) focus on higher-end micro-controllers with bundled costs of cloud services, which is not the norm. Most IoT vendors opt for inexpensive chips and platforms, with standalone firmware, which especially benefit from Capture's design.

10 Conclusion

Similar to other complex software systems, modern IoT devices suffer from the same security threats arising from poorly-managed outdated third-party libraries. We show that even the most popular smart device vendors fall behind the update schedules of critical libraries by hundreds of days, exposing users with even the latest device firmware to well-known vulnerabilities in the underlying libraries. These insights related to the usage of common third-party libraries across devices inspired the design of Capture, a software architecture for IoT firmware development. Capture provides mechanisms for centralized management of shared libraries by splitting functionality into the firmware on the device and a corresponding driver on a Capture Hub. Capture also provides strong isolation and security protections across devices and their drivers. Our evaluation results show that several example IoT devices can be modified to use Capture using one of our three integration approaches to get the security benefits of Capture with acceptable performance overheads.

Acknowledgement. We thank Jeremy Erickson, Haojian Jin and the members of the Synergy Labs for helpful discussions and feedback. Moreover, we greatly appreciate anonymous reviewers and our shepherd, Adwait Nadkarni, for their valuable comments and feedback. This work was partially supported by NSF Awards 1564009 and 1801472, Air Force Research Laboratory agreement number FA8750-15-2-0281, ONR Award N000141812618, and the Carnegie Mellon CyLab Security and Privacy Institute.

References

[1] Alasdair Allan. The problem with throwing away a smart device. https://www.hackster.io/news/the-problem-with-throwing-away-a-smart-device-75c8b35ee3c7, 2020.

[2] Alasdair Allan. Teardown of a smart plug (or two). https://www.hackster.io/news/teardown-of-a-smart-plug-or-two-6462bd2f275b, 2020.

[3] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. SoK: Security evaluation of home-based IoT deployments. In *2019 IEEE Symposium on Security and Privacy*, 2019.

[4] Amazon Web Services. AWS IoT Greengrass. https://aws.amazon.com/greengrass/, 2020.

[5] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the mirai botnet. In *26th USENIX Security Symposium*, 2017.

[6] Apache JMeter. https://jmeter.apache.org/, 2020.

[7] Andrew D Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 1984.

[8] Sudershan Boovaraghavan, Anurag Maravi, Prahaladha Mallela, and Yuvraj Agarwal. MLIoT: An end-to-end machine learning system for the Internet-of-Things. In *6th ACM/IEEE Conference on Internet of Things Design and Implementation*, IoTDI, 2021.

[9] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on Android. In *NDSS*, 2012.

[10] Daming D. Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for Linux-based embedded firmware. In *NDSS*, 2016.

[11] CNET. Search engine shodan knows where your toaster lives. https://www.cnet.com/how-to/shodan-dyn-botnet-searches-all-your-iot-devices/, 2015.

[12] Computer Weekly. Third-party code bug left instagram users at risk of account takeovers. https://www.computerweekly.com/news/252489542/Third-party-code-bug-left-Instagram-users-at-risk-of-account-takeover, 2020.

[13] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on Android. In *International Conference on Information Security*, 2010.

[14] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *15th International Conference on Mining Software Repositories*, MSR, 2018.

[15] Soteris Demetriou, Nan Zhang, Yeonjoon Lee, XiaoFeng Wang, Carl A. Gunter, Xiaoyong Zhou, and Michael Grace. HanGuard: SDN-driven protection of smart home WiFi devices from malicious mobile apps. In *10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec, 2017.

[16] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. Keep me updated: An empirical study of third-party library updatability on Android. In *2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2017.

[17] Brian Dipert. Teardown: A WiFi smart plug for home automation. https://www.edn.com/teardown-a-wi-fi-smart-plug-for-home-automation/, 2020.

[18] Brian Dipert. Teardown: WeMo switch is highly integrated. https://www.edn.com/teardown-wemo-switch-is-highly-integrated/, 2020.

[19] Colin Dixon, Ratul Mahajan, Sharad Agarwal, A.J. Brush, Bongshin Lee, Stefan Saroiu, and Paramvir Bahl. An operating system for the home. In *9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2012.

[20] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. Identifying open-source license violation and 1-day security risk at large scale. In *2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2017.

[21] Zakir Durumeric, David Adrian, Ariana Mirian, Michael Bailey, and J. Alex Halderman. A search engine backed by Internet-wide scanning. In *2015 ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2015.

[22] Jeremy Erickson, Qi Alfred Chen, Xiaochen Yu, Erinjen Lin, Robert Levy, and Z. Morley Mao. No one in the middle: Enabling network access control via transparent attribution. In *2018 ACM ASIA Conference on Computer and Communications Security*, AsiaCCS, 2018.

[23] ESP32.NET. The internet of things with ESP32. http://esp32.net/, 2020.

[24] Espressif. [SDK release] esp8266-nonos-sdk-v1.5.0-15-11-27. https://bbs.espressif.com/viewtopic.php?f=46&t=1442, 2015.

[25] Espressif. We just hit a new major milestone. https://www.espressif.com/en/media_overview/news/espressif-achieves-100-million-target-iot-chip-shipments, 2018.

[26] Espressif. ESP32 series datasheet. https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf, 2020.

[27] Espressif. WolfSSL for ESP-IDF. https://github.com/espressif/esp-wolfssl, 2020.

[28] Google. Protocol buffers. https://developers.google.com/protocol-buffers, 2020.

[29] gRPC — a high-performance, open source universal rpc framework. https://grpc.io/, 2020.

[30] Hang Guo and John Heidemann. Detecting IoT devices in the internet (extended). Technical Report ISI-TR-726B, USC/Information Sciences Institute, 2018.

[31] Hackaday.io. 382 projects tagged with "ESP32". https://hackaday.io/projects?tag=ESP32, 2020.

[32] Jun Han, Albert Jin Chung, Manal Kumar Sinha, Madhumitha Harishankar, Shijia Pan, Hae Young Noh, Pei Zhang, and Patrick Tague. Do you feel what I hear? enabling autonomous IoT device pairing using different sensor types. In *2018 IEEE Symposium on Security and Privacy*, 2018.

[33] Armijn Hemel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. Finding software license violations through binary code clone detection. In *8th Working Conference on Mining Software Repositories*, MSR, 2011.

[34] Tejun Heo. Control group v2. https://www.kernel.org/doc/Documentation/cgroup-v2.txt, 2015.

[35] David Hodson. Nest learning thermostat 2nd generation teardown. https://www.ifixit.com/Teardown/Nest+Learning+Thermostat+2nd+Generation+Teardown/13818, 2020.

[36] James Hong, Amit Levy, Laurynas Riliskis, and Philip Levis. Don't talk unless i say so! securing the Internet of things with default-off networking. In *3rd ACM/IEEE International Conference on Internet-of-Things Design and Implementation*, IoTDI, 2018.

[37] hostapd. https://w1.fi/hostapd/, 2020.

[38] IFTTT. https://www.ifttt.com, 2020.

[39] Ronny Ko and James Mickens. DeadBolt: Securing IoT deployments. In *Applied Networking Research Workshop*, ANRW, 2018.

[40] Deepak Kumar, Kelly Shen, Benton Case, Deepali Garg, Galina Alperovich, Dmitry Kuznetsov, Rajarshi Gupta, and Zakir Durumeric. All things considered: An analysis of IoT devices on home networks. In *28th USENIX Security Symposium*, 2019.

[41] Tobias Lauinger, Chaabane Abdelberi, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. Thou shalt not depend on me: Analysing the use of outdated JavaScript libraries on the web. In *NDSS*, 2017.

[42] The Security Ledger. Devices' UPnP service emerges as key threat to home IoT networks. https://securityledger.com/2019/03/devices-upnp-service-emerges-as-key-threat-to-home-iot-networks/, 2020.

[43] Xinyu Lei, Guan-Hua Tu, Chi-Yu Li, Tian Xie, and Mi Zhang. SecWIR: Securing smart home IoT communications via wi-fi routers with embedded intelligence. In *18th International Conference on Mobile Systems, Applications, and Services*, MobiSys, 2020.

[44] Chieh-Jan Mike Liang, Börje F. Karlsson, Nicholas D. Lane, Feng Zhao, Junbei Zhang, Zheyi Pan, Zhao Li, and Yong Yu. SIFT: Building an internet of safe things. In *14th International Conference on Information Processing in Sensor Networks*, IPSN, 2015.

[45] Linux containers. https://linuxcontainers.org/, 2020.

[46] Xianghang Mi, Feng Qian, Ying Zhang, and XiaoFeng Wang. An empirical characterization of IFTTT: ecosystem, usage, and performance. In *2017 Internet Measurement Conference*, IMC, 2017.

[47] MichMich. MagicMirror. https://github.com/MichMich/MagicMirror, 2020.

[48] Microsoft Azure. Azure Sphere. https://azure.microsoft.com/en-us/services/azure-sphere/, 2020.

[49] Asuka Nakajima, Takuya Watanabe, Eitaro Shioji, Mitsuaki Akiyama, and Maverick Woo. A pilot study on consumer IoT device vulnerability disclosure and patch release in japan and the united states. In *2019 ACM Asia Conference on Computer and Communications Security*, AsiaCCS, 2019.

[50] Hung Nguyen, Radoslav Ivanov, Linh T.X. Phan, Oleg Sokolsky, James Weimer, and Insup Lee. LogSafe: Secure and scalable data logger for IoT devices. In *3rd ACM/IEEE International Conference on Internet-of-Things Design and Implementation*, IoTDI, 2018.

[51] OpenSSL. Release strategy. https://www.openssl.org/policies/releasestrat.html, 2020.

[52] OpenSSL changelog. https://www.openssl.org/news/changelog.html, 2020.

[53] Particle device OS. https://www.particle.io/device-os/, 2020.

[54] phodal/awesome-iot. A collaborative list of great resources about IoT framework, library, OS, platform. https://github.com/phodal/awesome-iot#library, 2020.

[55] Raspberry Pi. Issue: cpuset disabled #1950. https://github.com/raspberrypi/linux/issues/1950, 2020.

[56] Raspberry Pi 4 specification. https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/, 2020.

[57] Dark Reading. Over 80% of medical imaging devices run on outdated operating systems. https://www.darkreading.com/iot/over-80--of-medical-imaging-devices-run-on-outdated-operating-systems/d/d-id/1337273, 2020.

[58] Red Hat. What is the maximum number of interface aliases supported in Red Hat Enterprise Linux? https://access.redhat.com/solutions/40500, 2020.

[59] Muzammil Abdul Rehman and Paul Grosu. RPC is not dead: Rise, fall and the rise of remote procedure calls. http://dist-prog-book.com/chapter/1/rpc.html, 2017.

[60] RTInsights. IoT devices still exposed, vast majority of traffic unencrypted. https://www.rtinsights.com/iot-security-remains-lacklustre/, 2020.

[61] RTInsights. Malware attacks IoT devices running windows 7. https://www.rtinsights.com/malware-iot-windows-7/, 2020.

[62] Samsung SmartThings. Direct-connected device SDK. https://smartthings.developer.samsung.com/docs/devices/direct-connected-devices/overview.html, 2020.

[63] Samsung SmartThings. SmartThings device SDK. https://github.com/SmartThingsCommunity/st-device-sdk-c, 2020.

[64] Senrio. 400,000 publicly available IoT devices vulnerable to single flaw. https://blog.senr.io/blog/400000-publicly-available-iot-devices-vulnerable-to-single-flaw, 2016.

[65] Mohit Sethi, Elena Oat, Mario Di Francesco, and Tuomas Aura. Secure bootstrapping of cloud-managed ubiquitous displays. In *2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp, 2014.

[66] Anna Kornfeld Simpson, Franziska Roesner, and Tadayoshi Kohno. Securing vulnerable home IoT devices with an in-hub security manager. In *2017 IEEE International Conference on Pervasive Computing and Communications Workshops*, PerCom, 2017.

[67] Andrew Stuart Tanenbaum and Robbert Van Renesse. *A critique of the remote procedure call paradigm*. 1987.

[68] TOMOYO linux. https://tomoyo.osdn.jp/index.html.en, 2020.

[69] smem(8) - linux man page. https://linux.die.net/man/8/smem, 2020.

[70] Rahmadi Trimananda, Ali Younis, Bojun Wang, Bin Xu, Brian Demsky, and Guoqing Xu. Vigilia: Securing smart home edge computing. In *2018 IEEE/ACM Symposium on Edge Computing*, SEC, 2018.

[71] Random Nerd Tutorials. 70+ ESP32 projects, tutorials and guides with Arduino IDE. https://randomnerdtutorials.com/projects-esp32/, 2020.

[72] Maarten Van Steen and Andrew S Tanenbaum. *Distributed systems*, chapter 4.2. 2017.

[73] Steve Vinoski. Convenience over correctness. *IEEE Internet Computing*, 12(4):89–92, 2008.

[74] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. In *International Workshop on Mobile Object Systems*, pages 49–64, 1996.

[75] Wikipedia. IEEE 802.11i-2004. https://en.wikipedia.org/wiki/IEEE_802.11i-2004, 2020.

[76] Wikipedia. OpenSSL - major version releases. https://en.wikipedia.org/wiki/OpenSSL#Major_version_releases, 2020.

[77] Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the Java system. 1996.

[78] Rich Wolski, Chandra Krintz, Fatih Bakir, Gareth George, and Wei-Tsung Lin. CSPOT: Portable, multi-scale functions-as-a-service for IoT. In *4th ACM/IEEE Symposium on Edge Computing*, SEC, 2019.

[79] Miguel G. Xavier, Marcelo V. Neves, Fabio D. Rossi, Tiago C. Ferreto, Timoteo Lange, and Cesar A. F. De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2013.

[80] Tianlong Yu, Vyas Sekar, Srinivasan Seshan, Yuvraj Agarwal, and Chenren Xu. Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the internet-of-things. In *14th ACM Workshop on Hot Topics in Networks*, HotNets, 2015.

[81] ZDNet. Shodan: The IoT search engine for watching sleeping kids and bedroom antics. https://www.zdnet.com/article/shodan-the-iot-search-engine-which-shows-us-sleeping-kids-and-how-we-throw-away-our-privacy/, 2016.

[82] ZDNet. CallStranger vulnerability lets attacks bypass security systems and scan LANs. https://www.zdnet.com/article/callstranger-vulnerability-lets-attacks-bypass-security-systems-and-scan-lans/, 2020.

[83] ZDNet. Hacker leaks passwords for more than 500,000 servers, routers, and IoT devices. https://www.zdnet.com/article/hacker-leaks-passwords-for-more-than-500000-servers-routers-and-iot-devices/, 2020.

[84] ZDNet. Ripple20 vulnerabilities will haunt the IoT landscape for years to come. https://www.zdnet.com/article/ripple20-vulnerabilities-will-haunt-the-iot-landscape-for-years-to-come/, 2020.

[85] Wei Zhang, Yan Meng, Yugeng Liu, Xiaokuan Zhang, Yinqian Zhang, and Haojin Zhu. HoMonit: Monitoring smart home apps from encrypted traffic. In *2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2018.

[86] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. FIRM-AFL: high-throughput greybox fuzzing of IoT firmware via augmented process emulation. In *28th USENIX Security Symposium*, 2019.