The Case for I/O-Device-as-a-Service

Ardalan Amiri Sani
University of California, Irvine
ardalan@uci.edu

Thomas Anderson
University of Washington
tom@cs.washington.edu

Abstract

Many computer systems, especially mobile and IoT systems, use a large number of I/O devices. A contemporary OS acts as a security guard for these devices, which trust the OS to correctly implement the "perimeter defense." Moreover, the OS also trusts these devices and their drivers to be well-behaved and bug-free. This interwoven trust model complicates the security of the system as a single vulnerable component can undermine all security guarantees. Not surprising, this architecture has failed to achieve strong security as evident by attacks that have targeted I/O devices or their drivers. In this paper, we call for a radically new approach, called *I/O-Device-as-a-Service (IDaaS)*, where each I/O device acts a separate service and is responsible for its own security. Inspired by Service-Oriented Architecture (SOA), IDaaS requires every device to be equipped with its own software stack and provide an externalizable API that can be safely exposed to untrusted software. We discuss the design decisions in IDaaS, highlight its security benefits and research challenges, and present a case study.

CCS Concepts • **Security and privacy → Systems security**; **Operating systems security**; **Mobile platform security**.

Keywords I/O devices; Service-Oriented Architecture

ACM Reference Format:
Ardalan Amiri Sani and Thomas Anderson. 2019. The Case for I/O-Device-as-a-Service. In *Workshop on Hot Topics in Operating Systems (HotOS '19), May 13–15, 2019, Bertinoro, Italy.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3317550.3321446

1 Introduction

A security design used quite often in practice is "perimeter defense." An intermediary interposes and vets all communications between untrusted parties and insecure components inside the perimeter. Often this is to allow for legacy components to be used securely, as with a firewall protecting against an external attacker exploiting a weak or out-of-date component. In this design, the security of the system depends on the thoroughness of the perimeter defense plus the operational security of every component inside the perimeter. In practice, this can be quite weak, as has been shown by the numerous security leaks of confidential enterprise information [7] or even a compromise of an automobile through a vulnerable telematics system [13].

OSes have long been built on this perimeter defense design: *I/O devices* attached to the computer depend on the OS, mainly the kernel, for their security. Any analysis of possible threats must include the device[1] itself, the driver software, the OS, and in most cases, other devices and their drivers. This is especially concerning for mobile and IoT systems due to their hardware diversity. For example, more than 24,000 distinct Android systems from more than 1000 manufacturers were available in the market in 2015 [29]. Such extreme diversity results in a large number of distinct I/O devices, such as cameras, audio devices, GPUs, various sensors such as accelerometer and compass, and several network devices such as WiFi, bluetooth, and NFC, each of which from several different manufacturers. The current OS security design means vulnerabilities in any of these devices or their drivers can result in devastating results.

Not surprisingly, this architecture has failed to provide strong security guarantees. First, the OS's trust on devices and their drivers is a continuous source of problems. Developed by third-party manufacturers, device drivers are a major source of crash bugs and security vulnerabilities in today's systems [15, 20, 41, 45]. For example, in 2016, 85% of kernel bugs in Android were in device drivers [41]. Even malicious device drivers have been spotted in the past, e.g., a driver used to implement a rootkit [35]. The device hardware is not necessarily trustworthy either and malicious devices can be used to attack the system, e.g., a malicious network device that leaks sensitive memory content. Second, I/O devices' trust in the OS is problematic too. Today, a compromise in the OS exposes all I/O devices, e.g., GPUs and disks, to untrusted applications. For example, a malicious application that compromises the OS can access all the buffers stored in the GPU even if those buffers belong to other applications.

Over the last several decades, OS and security researchers have proposed various solutions to alleviate the problems

[1]In the paper, we use the word "device" exclusively to refer to an I/O device.

caused by this design. Microkernels, exokernels, and user-space I/O frameworks execute device drivers outside the kernel [3, 9, 10, 17, 21, 25, 27, 31, 32, 34, 43]. Many solutions have also been proposed to deal with vulnerable device drivers in monolithic kernels, including in-kernel hardware-based and software-based sandboxing [12, 39], type-safety [16, 46], reference validation [42], automatic synthesis [36, 37], bug finding with static analysis [8, 18, 24, 30] and dynamic analysis [33, 38]. Other solutions have also been proposed to protect the system against malicious hardware devices [5].

However, these solutions address only part of the problem. For example, they only address the bugs in drivers or try to protect against a specific set of malicious devices. The sad result of lack of a comprehensive solution is simply today's *whack-a-mole* approach, where bugs and vulnerabilities are patched when found (or worse when exploited).

In this paper, we call for a radically new approach: *all I/O devices must be designed and integrated with the system as mutually distrusting services*. Our proposal is inspired by Service-Oriented Architecture (SOA), in which every service in a datacenter is responsible for its own operation with an external interface open to untrusted use. In other words, such a service can be made visible as an API to the outside world without depending on a perimeter defense system. We refer to this proposed architecture as *I/O-Device-as-a-Service* (*IDaaS*). IDaaS addresses the core of the security problems regarding I/O devices: the mutual trust between the OS and these devices. In IDaaS, all services are mutually distrusting – neither the OS nor the device are trusted by the other.

The IDaaS architecture advocates a key principle: *all I/O devices should be responsible for their own security*. This principles has several design implications. First, it means the API provided by an I/O device must be *externalizable*. That is, the OS should be able to securely expose this API to untrusted applications. Obviously, this API must be *narrow and well-defined*. In §4, we provide an example of externalizable API for camera. Second, to implement such an API, I/O devices must include their own software stack (including device drivers). I/O devices today provide a low-level hardware interface, such as registers and interrupts, to the OS. The OS then uses a device driver as well as some libraries and user space daemons to implement higher-level logical APIs for applications. In IDaaS, the device itself must run all the required software stack (e.g., device drivers, libraries, and daemons) on an internal microcontroller in order to directly implement the logical API. Finally, I/O devices must implement their own security mechanisms and not rely on the OS. This includes secure boot, data isolation, and authentication (§3.3).

Indeed, some I/O devices already adhere to some of these principles. First, some devices implement their own security mechanisms. For example, SR-IOV devices support virtual modes, which untrusted users or virtual machines can directly access [26]. Second, some devices run part of the software stack on microcontrollers (rather than in the OS). For example, the Imaging SubSystem (ISS) in OMAP4 mobile SoC leverages Cortex M3 microcontrollers to execute its firmware, which directly interfaces with the camera, communicates with the main processor using a message-passing interface, and delivers the frames to the OS by copying them into main memory [40]. However, none of these devices adhere to all the principles needed to fully decouple I/O devices and the OS.

Eliminating the trust between the OS and devices has an important design implication. That is, an application cannot rely on the OS for secure communication with a device; it must use a secure enclave instead to establish a secure channel. Without enclaves, IDaaS still provides half of its benefits by isolating the device and its software from the OS. It cannot however eliminate trust on the OS. Note that IDaaS can use various realizations of secure enclaves, e.g., Intel SGX enclaves, trusted applications running in ARM TrustZone secure world, or applications protected from the OS by a more privileged layer, e.g., hypervisor [14, 23].

Pushing the security burden to I/O devices might seem counter-intuitive and one might wonder how such a design can improve the overall security of the system. We see four fundamental reasons. First, as device drivers and devices are isolated from the kernel, their vulnerabilities (and even malice) do not lead to kernel exploits. In IDaaS, vulnerabilities in the device software stack can only lead to attacks directed at that specific I/O device, e.g., phishing attacks through the compromise of the display. Such attacks are more limited in scope. Note that microkernels also isolate device drivers from the kernel, but they do not provide the rest of the security benefits of IDaaS.

Second, the externalizable interface of I/O devices is narrow, hence the attack surface on these devices is much smaller. This means that the same vulnerabilities that can be exploited through the wide attack surface of a kernel device driver might not be exploitable under the narrow interface of IDaaS.

Third, as devices do not rely on the OS for their security, a compromise in the OS does not automatically lead to the compromise of these devices.

Finally, quite counter-intuitively, we believe that IDaaS architecture can improve the quality of the software stack of I/O devices developed by their manufacturers. We see two reasons for this. First, IDaaS has software engineering benefits – it allows the manufacturer to develop and test only a single version of the software stack (rather than multiple versions for different OSes or different OS versions), which helps it focus its resources. Second, IDaaS helps with blame allocation – if an I/O device gets compromised, the manufacturer will be exclusively blamed. This will motivate the device manufacturer to improve the quality of its software.

One might think that IDaaS will require manufacturers to write more code (which may result in more bugs and vulnerabilities). We expect the extra code to be small including some code to boot the device and to implement an interface to receive device requests. Moreover, we expect this extra code to be reused across various devices allowing it to be properly tested.

In addition to enhancing the system's security, IDaaS will have three other benefits. First, IDaaS may facilitate innovation in OS design since porting device drivers is a barrier for adoption of new OSes. Second, IDaaS allows the OS and device software to be upgraded separately. Third, IDaaS may reduce the energy consumption of the system by shifting work from power-hungry processors to more efficient microcontrollers.

The IDaaS architecture requires a microcontroller and a small amount of memory for every I/O device. We believe this is not a difficult requirement to satisfy as (*i*) many modern I/O devices already have microcontrollers [19], (*ii*) the required amount of hardware resources to meet this goal is small, and (*iii*) hardware is increasingly cheap.

2 Background

Our proposed architecture is inspired by the success of Service-Oriented Architecture (SOA) in revolutionizing cloud computing. As an example, around 2002, Amazon decided to redesign itself using SOA [44]. That is, it decided that every functionality and data in Amazon must be provided through services with externalizable APIs. This design forbade cross-talk and direct links between various components. All exchanges had to be implemented through the service APIs. Amazon's decision to adopt SOA meant that each service had to provide a well-defined API for other services and clients to interact with it. This also meant that each service had to be designed with security in mind as each service was now exposed, through its API, to the outside world.

An SOA-based design can improve the overall security of a system for one key reason: it reduces the complexity of the system, making it easier to reason about and implement security mechanisms. In contrast, in a highly integrated system, different components interact through complex, not-well-documented, and hard-to-analyze interfaces. This is problematic as the reliability and security of every component now relies on other components. It also requires teams in charge of various components to remain in constant communication, e.g., for vulnerability patching, which further complicates development.

3 Overview

Inspired by SOA, we propose a radical architecture, *I/O-Device-as-a-Service* (*IDaaS*), for the support of I/O devices in a computer system. We propose that every I/O device must act as an independent service. It must not trust the OS; instead, it must be fully responsible for its own security and provide its own externalizable API. We expect each I/O device to be equipped with a microcontroller and a small amount of memory to run the device driver and the rest of the software stack for the device. Moreover, in this design, the kernel does not trust the device and its software either. Figure 1 illustrates the existing and proposed architectures.

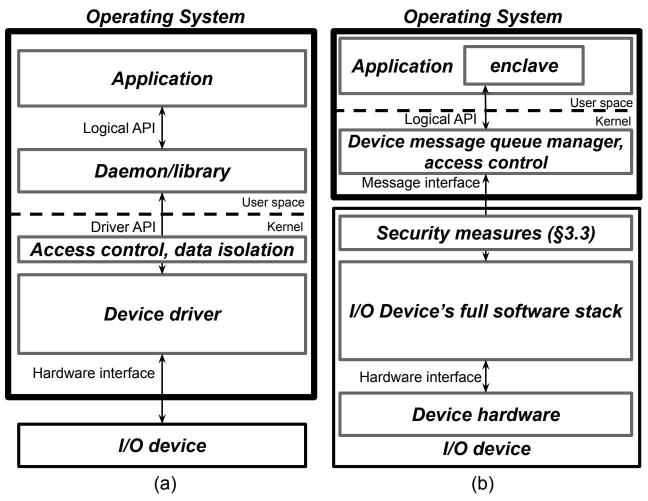

Figure 1. *(a) Current architecture. (b) Proposed SOA-based IDaaS architecture.*

One key aspect of IDaaS is the decoupling of the device and its software from the OS. This makes our proposal different from existing work that refactors an I/O device software stack for performance reasons. For example, Helios runs a satellite kernel on every programmable device, such as a network device [28]. The goal of Helios is to improve the performance by running the application closer to the hardware it uses. However, it still requires the OS to provide device drivers for the device. Moreover, Helios assumes that the device fully trusts the OS. For example, Helios requires the OS to load the software running on the programmable device. In IDaaS, devices do not trust the OS. They do implement their own security mechanisms, e.g., secure boot to guarantee the right software is loaded on the device.

IDaaS may be applied to any peripheral device in the system. However, we mainly target IDaaS for I/O devices used by applications, such as camera, GPU, audio devices, display, touchscreen, sensors, network devices, and storage devices. For simpler peripheral devices that are not directly used by applications, such as a clock or a voltage regulator, one can use a simpler model – either the current fully-trusted model or a model where the device trusts the kernel but the kernel does not trust the device and limits its behavior.

3.1 Externalizable API

In IDaaS, the API of an I/O device must be externalizable. That is, the API must be safely exposed to untrusted applications. This has two important implications. First, the APIs provided by I/O devices must be high-level logical APIs. Today's I/O devices expose to the kernel a low-level hardware interface (i.e., registers and interrupts), which cannot be safely exposed to untrusted applications. Even the drivers do not typically provide such logical APIs. Instead, they expose a large number of custom `ioctl` syscalls that are either only exposed to OS daemons in user space (since they cannot be safely exposed to untrusted applications) or are exposed to untrusted applications resulting in a wide attack surface. However, high-level logical APIs for I/O devices often exist. They are typically provided by user space libraries and daemons. As an example, the camera service process in Android provides a camera API for applications (available through Binder IPC calls). In our proposed architecture, we argue that I/O devices should directly provide the high-level logical API used by applications. This means that the device must include and run the whole software stack (including the device driver, libraries, and daemons) on a microcontroller on the device. In addition, we suggest that the API should only include a few high-level calls with clear semantics. This has the important advantage of reducing the attack surface on the device software.

Second, in IDaaS, an I/O device does not provide a privileged interface for calls from the kernel (unlike an SR-IOV device that has a privileged interface for the kernel as well as virtual interfaces for untrusted applications or virtual machines). It only exposes a single set of APIs, which can be called by any software.

3.2 Message Queue-based Interface

In IDaaS, the interface to every device is a message queue. Software can insert API call requests on the queue and the I/O device services these requests. Queues are a common primitive in message-passing microkernels. We generalize this primitive to devices as well.

The default design decision in IDaaS is for each I/O device to provide a single message queue, which is managed by the OS. This means that the OS mediates applications' accesses to device queues. This might cause performance overhead, which can be an issue for devices with high performance requirement. To address this problem, high performance devices can optionally provide multiple queues, each of which can be directly mapped into an application's address space to enable direct access. If needed, performance can be further improved by using per-application per-core queues. This is similar to SR-IOV devices. However, IDaaS is different from SR-IOV as it does not have a privileged interface for the OS to program and configure the device and hence does not require a device driver in the OS.

In IDaaS, the role of the OS is to implement access control. That is, the OS only *grants or deny applications access to devices' message queues* (see the queue manager depicted in Figure 1). This design has an important implication on the threat model of the system: in the presence of a malicious kernel, IDaaS will be able to provide integrity and confidentiality guarantees for the data produced, stored, or used by I/O devices, but cannot guarantee the availability of the device. For example, a display controller can protect the confidentiality of the data shown to the user (similar to SchrodinText [4]) and a GPU can protect the data buffers sent to it by applications for 3D rendering. But these devices cannot guarantee that they will be available to service applications' requests when needed. We believe this is an acceptable threat model as a compromised kernel has many other ways to mount availability attacks on applications, e.g., refusing to execute them.

Note that since the OS enforces access control, it can allow any app, including malicious ones, to use an I/O device. For example, it might allow a malicious application to eavesdrop on the user through the camera. Addressing these other types of attacks is out of the scope of IDaaS.

3.3 Security Measures

Every I/O device in IDaaS must implement its own security measures including:

1. Secure boot. Every device should check the integrity and authenticity of its software image at load time rather than trusting the kernel. This design can prevent attacks that attempt to deploy malicious firmware on an I/O device [11]. Moreover, secure boot prevents all device software upgrades other than those by the manufacturer.

Secure boot requires the device to have the public key of its manufacturer in secure persistent storage. To achieve this, the device should use some form of a read-only memory (e.g., "One Time Programmable (OTP) or eFuse memory" [6]) only available to it.

2. Data isolation. Those devices that store sensitive information of different applications must provide isolation mechanisms to protect the data, e.g., through paging. Some devices already support such isolations mechanisms, e.g., GPUs. However, today's devices rely on the OS-based driver to program the isolation-related resources, e.g., device page tables (which translate device virtual addresses to physical addresses). In IDaaS, these resources must be configured directly by the on-device software.

3. Authentication & secure communication. Given the externalizable API of I/O devices, applications can directly communicate with them. This raises important challenges: authenticating the requests and protecting the confidentiality and integrity of communication. To address these challenges, applications and I/O devices should use secure channels for communication. To bootstrap such a channel, an I/O device should have a persistent device key (i.e., a private key uniquely available to the device).

We note that cryptographic operations add performance overhead. Therefore, we suggest using them judiciously. For example, if the confidentiality of the data is not important, encryption should be avoided. We suggest leaving this decision to applications and devices.

3.4 Enclaves for Secure I/O

Since the OS is not trusted in IDaaS, applications should use secure enclaves to communicate with devices. In today's systems, applications trust the OS to be able to impersonate them, e.g., by programming I/O devices on their behalf. And in that setting, the device must also trust the user's delegation to the OS. IDaaS supports a stricter model, where the OS does not need to be trusted. That is, an application can use a hardware-backed enclave to establish a secure communication channel with an I/O device without relying on the OS. IDaaS can use various realizations of secure enclaves, e.g., Intel SGX enclaves, trusted applications running in ARM TrustZone secure world, or applications protected from the OS by a more privileged layer, e.g., hypervisor [14, 23]. Note that without enclaves, IDaaS still provides half of its benefits by isolating the device and its software from the OS. It cannot however eliminate trust on the OS.

Existing systems have already advocated for this model. For example, Graviton, which targets trusted execution on GPUs, enables an application to create a secure communication channel with the GPU to exchange sensitive data without trusting the OS. IDaaS extends this to other I/O devices as well. It also requires I/O devices to implement additional security features, such as secure boot. Moreover, unlike Graviton, IDaaS requires redesigning of the APIs exposed by I/O devices to be externalizable.

4 A Case Study

In this section, we discuss how a camera can adopt IDaaS. Current device drivers for cameras implement a large set of `ioctl` syscalls. For example, the Qualcomm MSM camera device driver used in many mobile systems, such as Nexus 5X and Nexus 6P, consists of about 65,000 kernel LoC and implements about 120 `ioctls`, resulting in a large trusted computing base (TCB) with a wide attack surface. This driver implements various low-level functionalities of the camera including image capture, frame streaming, image processing (e.g., flipping, rotating, denoising, and cropping), compression, and flash. It is no surprise that this single driver has so far been the host of several bugs and vulnerabilities, which can even be exploited for kernel code injection [38].

Instead, we suggest that the camera should include the software needed to operate the camera and expose mainly a single externalizable API call: `capture_frame(conf, buf, sec_ops)`. With this API design, an application (permitted to access the camera by the OS) can receive a camera frame by sending a single `capture_frame` message to the camera device. The first parameter of the message is the set of configuration options required for that specific frame including resolution, pixel format, flash option, and lens focus. The second parameter is the buffer in the application's address space where the frame needs to be written. Finally, the last parameter is an optional one. If provided, it asks the camera to sign or encrypt the frame before storing it into the buffer. This single API call can support video capture as well, in which case the application sends consecutive `capture_frame` messages to the camera.

In fact, a similar camera API has been recently introduced in Android. In Android, the Hardware Abstraction Layer (HAL) in user space implements APIs for various types of I/O devices. HAL version 3 of camera has adopted a similar message-based API (although it does not support signed or encrypted frames) [22]. This demonstrates the feasibility of using a narrow API for an I/O device. However, note that the Android HAL runs in a daemon process in user space, still requiring a large and complex device driver in the kernel. Moreover, in today's system, the camera device must trust the OS. Our proposal is to move the software stack to the camera itself and eliminate the trust between the camera and the OS.

While we only elaborate on one case study here, one can envision similar APIs for other I/O devices. For example, an audio device can provide a single API to record or play an audio segment of an adjustable length. A sensor can implement a single API to capture a reading. Even a complex device such as GPU can be used with a small number of APIs. For example, a GPU can expose an API that accepts a shader kernel and all its inputs at once, performs the computation and rendering, and returns the output results and buffers to the caller. In contrast, in today's systems, these devices expose a complex hardware interface to the OS, and their device drivers exposes tens of `ioctl` syscalls to user space.

5 Research Challenges & Discussions

5.1 Peripheral Buses

In today's systems, I/O devices are typically connected to the system bus through peripheral buses, e.g., I^2C and PCI. These buses facilitate the connection of diverse – often weak – hardware devices to different system buses operating at high frequencies. In order to use a device connected to the bus, the OS needs to program the bus using a bus driver. The buses are diverse, hence there are many drivers for them. For example, the latest stable Linux kernel version at the time of writing (version 4.20.2) contains drivers for more than 100 different I^2C buses, collectively more than 50,000 LoC. What happens to these bus drivers in IDaaS?

We argue that the OS should not contain custom bus drivers as it violates the IDaaS principle of decoupling of I/O devices (and their buses) from the OS. Instead, the bus driver should run on a microcontroller closer to the bus. We see two cases. First, for an I/O device that is integrated in the system and is the sole user of the bus, the same microcontroller that runs the device software stack can run the bus driver. This microcontroller connects to the peripheral bus (which itself connects to the device hardware). It also communicates with the main processor (which executes the OS).

Second, for external devices, e.g., USB devices, and for those sharing a bus, a separate microcontroller dedicated to the bus can be used. These I/O devices still need to have their full software stack running on their own microcontrollers. In contrast to the previous case, however, the peripheral bus here has its own microcontroller, which runs the bus driver and interfaces with the main processor. This bus microcontroller forwards the messages from the main processor to the I/O device and vice versa.

For the device and bus microcontrollers' communication with the main processor, we recommend a hardware communication medium that provides some memory space and interrupts for exchanging messages. A good example of such a medium is the hardware mailbox used in OMAP SoCs for communication between various microprocessors, microcontrollers, and accelerators, e.g., Cortex A9 microprocessors, Cortex M3 microcontrollers, and a DSP in OMAP4 [40]. This mailbox provides interrupts and message queues with a limited number of queues and entries in each queue.

5.2 Secure Memory Access

Some I/O devices may need to use DMA for exchanging data with applications. However, DMA results in a large attack surface as it requires trusting the driver (to set up the DMA correctly) and the device (to obey the DMA instructions).

We discuss two design decisions to address the challenges with DMA in IDaaS. First, simpler I/O devices should not use DMA at all. Many simple sensors and actuators do not require exchanging large amounts of data with applications. In such cases, the data can be directly exchanged. For example, OMAP4 hardware mailbox supports 32-bit messages [40], which can easily carry the data of simple I/O devices.

Second, I/O devices requiring DMA for large amounts of data (e.g., camera images and GPU buffers) must be protected by I/O Memory Management Units (IOMMUs). That is, the kernel must program the IOMMUs to limit the DMA targets to only part of the corresponding application's memory, which is used to exchange data with the I/O device. Unrestricted DMA access to application's memory, or even worse, to the system memory, must be prohibited. If not, the device will have the ability to overwrite critical memory regions, e.g., the kernel memory or an app's code section. When using enclaves, the memory buffer accessed by DMA will be a bounce buffer sitting outside the enclave memory, requiring signing or encryption for security. Note that this design means that a malicious kernel can allow a device to overwrite an application's memory (but not the enclave memory). This is not a new attack vector as the kernel can directly write to an application memory.

5.3 Economy of IDaaS

We believe that the device software can run on a weak or – for some complex devices – a moderately powerful microcontroller on the device. Yet, one might wonder if requiring one microcontroller per I/O device might be too expensive in practice. We believe that this requirement can be met. First, many I/O devices already have microcontrollers, e.g., GPUs [19] and cameras [40]. Second, today's mobile SoC's incorporate billions of transistors (e.g., 7 billion and 10 billion for Apple A12 and A12X Bionic, respectively [1, 2]). A small microcontroller requires anywhere between 10s of thousands to a few millions of transistors depending on its features. As a result, using one microcontroller for a few tens of I/O devices requires a small fraction of the overall transistor budget on a chip.

Moreover, we see two opportunities for sharing. The first one is sharing a microcontroller between devices from the same manufacturer. In an SoC, often a few devices are from the same manufacturer. For example, it is common to see Qualcomm's GPU and camera on a device with a Qualcomm SoC. In this case, these I/O devices can share a microcontroller. Similarly, several sensors might be from the same manufacturer. While this design breaks the strong IDaaS isolation between the sharing devices, we think it is acceptable since all of them are from the same manufacturer and trust each other.

The second opportunity is for sharing a security co-processor. In IDaaS, I/O devices require cryptographic operations to implement their security measures (§3.3). While a software implementation is feasible, using a security co-processor can help with performance. However, requiring a security co-processor for all I/O devices, especially for simpler ones, might not be economic. Therefore, for simpler devices, one can consider sharing a security co-processor. Such sharing must be done carefully to provide strong isolation between sharing devices and to protect the co-processor from the OS.

6 Conclusions

We introduce IDaaS, a system architecture for secure integration of I/O devices in a computer system. In this architecture, each I/O device is a service, which runs its own software stack, is in charge of its own security, and does not trust the OS for perimeter defense. The OS does not trust the device and its software stack either. This results in reduced complexity and has the potential to improve the overall system security, but poses a number of challenges for future research.

Acknowledgments

This work is supported in part by NSF Awards #1617513 and #1518702 as well as gifts from Google, Facebook, and Huawei. The authors thank Zhihao Yao, Kevin Zhao, and the anonymous HotOS reviewers for their valuable feedback on the ideas proposed in this paper.

References

[1] A12 Bionic - Apple. https://en.wikichip.org/wiki/apple/ax/a12, 2019.

[2] A12X Bionic - Apple. https://en.wikichip.org/wiki/apple/ax/a12x, 2019.

[3] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation For UNIX Development. In *Proc. Summer 1986 USENIX Conference*, 1986.

[4] A. Amiri Sani. SchrodinText: Strong Protection of Sensitive Textual Content of Mobile Applications. In *Proc. ACM MobiSys*, 2017.

[5] S. Angel, R. S. Wahby, M. Howald, J. B. Leners, M. Spilo, Z. Sun, A. J. Blumberg, and M. Walfish. Defending against Malicious Peripherals with Cinch. In *Proc. USENIX Security Symposium*, 2016.

[6] ARM. Juno ARM Development Platform SoC, Revision r0p0, Technical Overview. *ARM DTO*, 0038A (ID040516), 2014.

[7] T. Armerding. The 17 biggest data breaches of the 21st century. https://www.csoonline.com/article/2130877/data-breach/the-biggest-data-breaches-of-the-21st-century.html, 2018.

[8] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough Static Analysis of Device Drivers. In *Proc. ACM EuroSys*, 2006.

[9] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proc. USENIX OSDI*, 2014.

[10] S. Boyd-Wickizer and N. Zeldovich. Tolerating Malicious Device Drivers in Linux. In *Proc. USENIX ATC*, 2010.

[11] M. Brocker and S. Checkoway. iSeeYou: Disabling the MacBook Webcam Indicator LED. In *Proc. USENIX Security Symposium*, 2014.

[12] M. Castro, M. Costa, J. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast Byte-granularity Software Fault Isolation. In *Proc. ACM SOSP*, 2009.

[13] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno. Comprehensive Experimental Analyses of Automotive Attack Surfaces. In *Proc. USENIX Security Symposium*, 2011.

[14] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports. Overshadow: a Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proc. ACM ASPLOS*, 2008.

[15] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating Systems Errors. In *Proc. ACM SOSP*, 2001.

[16] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the Real World. In *Proc. ACM PLDI*, 2003.

[17] D. R. Engler, M. F. Kaashoek, and J. O. Jr. Exokernel: an Operating System Architecture for Application-Level Resource Management. In *Proc. ACM SOSP*, 1995.

[18] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software Guards for System Address Spaces. In *Proc. USENIX OSDI*, 2006.

[19] Y. Fujii, T. Azumi, N. Nishio, and S. Kato. Exploring Microcontrollers in GPUs. In *Proc. ACM Asia-Pacific Workshop on Systems (APSys)*, 2013.

[20] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows XP Kernel Crash Analysis. In *Proc. USENIX LISA*, 2006.

[21] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The Design and Implementation of Microdrivers. In *Proc. ACM ASPLOS*, 2008.

[22] Google. Camera HAL3 in Android. https://source.android.com/devices/camera/camera3, 2018.

[23] J. Hoffmann, M. Ussath, T. Holz, and M. Spreitzenbarth. Slicing droids: Program slicing for smali code. In *Proc. ACM Symp. Applied Computing (SAC)*, 2013.

[24] A. Kadav, M. J. Renzelmann, and M. M. Swift. Tolerating Hardware Device Failures in Software. In *Proc. ACM SOSP*, 2009.

[25] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Proc. ACM SOSP*, 2009.

[26] P. Kutch. PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology. *Intel Application Note*, 321211-002, Revision 2.5, 2011.

[27] J. Liedtke. Improving IPC by Kernel Design. *ACM SIGOPS Operating Systems Review*, 1993.

[28] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In *Proc. ACM SOSP*, 2009.

[29] OpenSignal. ANDROID FRAGMENTATION VISUALIZED (AUGUST 2015). https://www.opensignal.com/sites/opensignal-com/files/data/reports/global/data-2015-08/2015_08_fragmentation_report.pdf.

[30] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and Automating Collateral Evolutions in Linux Device Drivers. In *Proc. ACM EuroSys*, 2008.

[31] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The Operating System is the Control Plane. In *Proc. USENIX OSDI*, 2014.

[32] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the Library OS from the Top Down. In *Proc. ACM ASPLOS*, 2011.

[33] M. J. Renzelmann, A. Kadav, and M. M. Swift. SymDrive: Testing Drivers without Devices. In *Proc. USENIX OSDI*, 2012.

[34] M. J. Renzelmann and M. M. Swift. Decaf: Moving Device Drivers to a Modern Language. In *USENIX ATC*, 2009.

[35] M. Russinovich. Sony, Rootkits and Digital Rights Management Gone Too Far. https://blogs.technet.microsoft.com/markrussinovich/2005/10/31/sony-rootkits-and-digital-rights-management-gone-too-far/, 2005.

[36] L. Ryzhyk, P. Chubb, I. Kuz, E. Le Sueur, and G. Heiser. Automatic Device Driver Synthesis with Termite. In *Proc. ACM SOSP*, 2009.

[37] L. Ryzhyk, A. Walker, J. Keys, A. Legg, A. Raghunath, M. Stumm, and M. Vij. User-Guided Device Driver Synthesis. In *Proc. USENIX OSDI*, 2014.

[38] S. M. Seyed Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. Amiri Sani, and Z. Qian. Charm: Facilitating Dynamic Analysis of Device Drivers of Mobile Systems. In *Proc. USENIX Security Symposium*, 2018.

[39] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proc. ACM SOSP*, 2003.

[40] Texas Instruments. Architecture Reference Manual, OMAP4430 Multimedia Device Silicon Revision 2.x. SWPU231N, 2010.

[41] J. Vander Stoep. Android: Protecting the Kernel. In *Linux Security Summit (LSS)*, 2016.

[42] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device Driver Safety Through a Reference Validation Mechanism. In *Proc. USENIX OSDI*, 2008.

[43] Z. Yao, Z. Ma, Y. Liu, A. Amiri Sani, and A. Chandramowlishwaran. Sugar: Secure GPU Acceleration in Web Browsers. In *Proc. ACM ASPLOS*, 2018.

[44] S. Yegge. Stevey's Google Platforms Rant. https://gist.github.com/chitchcock/1281611, 2011.

[45] H. Zhang, D. She, and Z. Qian. Android Root and its Providers: A double-Edged Sword. In *Proc. ACM CCS*, 2015.

[46] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In *Proc. USENIX OSDI*, 2006.