Secure I/O on Trusted Platforms with Lightweight Kernels

by

Nicholas Gordon

Bachelor of Science, University of Memphis, 2016

Submitted to the Graduate Faculty of

the Department of Computer Science in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2024

UNIVERSITY OF PITTSBURGH

DEPARTMENT OF COMPUTER SCIENCE

This dissertation was presented

by

Nicholas Gordon

It was defended on

August 8, 2024

and approved by

Dr. Amy Babay, DINS and Department of Computer Science, University of Pittsburgh

Dr. Adam Lee, Department of Computer Science, University of Pittsburgh

Dr. Devesh Tiwari, Department of Computer Science, Northeastern University

Dissertation Director: Dr. John Lange, Department of Computer Science, University of Pittsburgh

Secure I/O on Trusted Platforms with Lightweight Kernels

Nicholas Gordon, PhD

University of Pittsburgh, 2024

Trusted computing has become widespread and the complexity of trusted applications has increased substantially, such as in real-time patient vitals data processing or employee-free stores that continuously monitor customers. These applications differ from existing trusted computing usage in that they directly acquire and process sensitive information from sensors like cameras and microphones. Simultaneously, application demands are expanding to include a rich, general-purpose OS environment to provide network, filesystems, and multicomputing capabilities. An application runtime of similar capability approaches an OS in terms of complexity and would require extensive interfacing with the underlying untrusted OS anyway, so we claim that a full-stack trusted OS provides similar capabilities with a smaller, less complex trust profile. Further, current trusted OSes fail to provide this environment because they are designed to provide trusted *services* to untrusted applications, and the use of full-weight kernels (FWKs) like Linux is ruled out due to security concerns. We aim to solve this problem by using lightweight kernels (LWK), which strike the correct balance between security and usability and can fully exploit hardware to provide secure device I/O.

Lightweight kernels are an OS design approach that presents a familiar programming environment to Linux both in userspace and in the kernel, allowing many applications to run without modification, as well as ease porting of existing device drivers. Further, hardware is more directly exposed to programmers–that is, with fewer hardware abstraction layers–enabling easy leveraging of platform hardware and peripherals. To demonstrate these design advantages we develop a LWK trusted OS for the ARM TrustZone environment on a typical IoT or edge computing hardware platform. Specifically, we extend the Kitten LWK to be TrustZone-aware, develop an I/O stack to demonstrate the viability of a camera driver, and then build a framework for securely paravirtualizing existing Linux drivers by using recent, modern TrustZone hardware.

Table of Contents

Preface . . . xi

1.0 Introduction . . . 1

1.1 Motivation: Autonomous/Smart Shopping . . . 5

1.2 Thesis . . . 7

1.3 Research Contributions and Insights . . . 8

1.3.1 Trusted Execution Environment (TEE) . . . 8

1.3.2 Secure Peripheral Device I/O . . . 9

1.3.3 Driver Paravirtualization . . . 9

2.0 Threat Model . . . 11

2.1 Attestation . . . 14

3.0 LWKs as Secure OSes . . . 15

3.1 Background and Related Work . . . 15

3.1.1 Trusted Execution Environment Models . . . 15

3.1.2 System Software Assumptions and Application APIs . . . 16

3.1.2.1 Microkernels and Unikernels . . . 19

3.1.3 TEE OSes and Software Runtimes . . . 20

3.1.3.1 Runtime Systems . . . 21

3.1.3.2 Service-like OSes . . . 21

3.1.3.3 Formal Verification . . . 24

3.1.3.4 Existing Mainstream Kernels . . . 24

3.1.3.5 Commercial OSes . . . 25

3.1.4 Lightweight Kernels . . . 25

3.2 LWKs as TEE OSes . . . 27

3.2.1 Hardware Assumptions . . . 28

3.2.2 System Co-existence . . . 28

3.2.3 Partition Boundaries and Messaging: the Interkernel . . . 30

3.3 KaTZe: The Kitten LWK TEE OS 31
3.3.1 Resource efficiency and TCB simplicity 32
3.3.2 Secure Interrupts 33
3.3.3 The Interkernel Channel 35
3.3.3.1 Sockets 37
3.4 Evaluation 38
3.4.1 TrustZone Performance 38
3.4.2 Application Support and Environment Familiarity 38
3.4.2.1 Mongoose Webserver 40
3.4.2.2 SQLite3 Database 40
3.4.2.3 SOD 41
3.4.2.4 Mmap-based IPC 41
3.4.2.5 WebAssembly Runtime: wasm3 42
3.4.3 Prototype Application and Benchmarks 42
3.4.3.1 SQLite3 speedtest1 43
3.4.3.2 Mongoose performance: sockets 44
3.4.4 Conclusion 44
4.0 Secure I/O Stack with Lightweight Kernels 46
4.1 Background and Related Work 46
4.1.1 Protecting the data 47
4.1.2 Protecting the channel 48
4.2 Trusting I/O: Secure Driver Stacks 50
4.2.1 Devices and internal complexity 52
4.3 Simplifying the Driver Stack 53
4.3.1 Driver Complexity: Video4linux2 and the Media Controller Framework 55
4.4 KaTZe Implementation of Trusted I/O Devices 56
4.4.1 Secure Device Access 56
4.4.2 Underlying System Bus Architecture 56
4.4.3 HTU21D Sensor: I2C Control and Data 57

4.4.4 IMX214 Camera: I2C Control, D-PHY Data . . . 58
4.4.5 SoC Device Complexity and Documentation: RK3399 ISP . . . 58
4.4.6 Limitation: Undocumented Security Features . . . 60
4.4.7 Userland Camera Interface . . . 61
4.5 Evaluation . . . 62
4.5.1 ISP TrustZone Overhead . . . 62
4.5.2 Image Recognition on Captured Frames . . . 63
4.5.3 Conclusion . . . 64
5.0 Paravirtual Device Drivers . . . 65
5.1 Security Sensitivity of Devices . . . 66
5.2 Background and Related Work . . . 68
5.3 Split Drivers to Reduce Trusted TCB . . . 70
5.3.1 The "Composite" Devices of the Paravirtual Framework . . . 71
5.3.2 Containing Platform Complexity . . . 72
5.4 Implementation of Split Drivers in KaTZe . . . 74
5.4.1 Privacy and Security Considerations . . . 74
5.4.2 Re-Using Existing Linux Configurations . . . 75
5.4.3 Device Limitations: Memory Access Granularity . . . 77
5.5 Evaluation . . . 78
5.5.1 HTU21D: Paravirtual vs Ported . . . 78
5.5.2 TCB Reduction . . . 79
5.5.3 Conclusion . . . 80
6.0 Discussion . . . 82
6.1 Support for Attestation Mechanisms . . . 82
6.2 Conclusion and Further Work . . . 84
Bibliography . . . 85

List of Tables

1 LWKs are comparable to existing TEE OSes in terms of code complexity. . . . 27

2 TCB measurements for trusted kernel candidates. 33

3 Paravirtualizing the platform drivers results in an average reduction of ~57% per device, or an overall LOC reduction of 53%. 81

List of Figures

1 Processing data at the sensor and separated management and data access planes are key components of our target class of applications. 2
2 Our threat model ensures that secure partition software cannot be tampered with by the owner. 12
3 The GIC's hierarchical structure means the two partitions may compete for control of the GIC distributor's configuration. Diagram from ARM [7] 34
4 The interkernel provides a generic message-passing system to implement arbitrary services on top. 36
5 KaTZe provides socket operations by delegating them to the Linux kernel. . . . 37
6 TrustZone memory protection imposes no overhead in the STREAM benchmarks. 39
7 TrustZone memory protection imposes no overhead in the RandomAccess (GUPS) benchmarks. 39
8 KaTZe supports a typical POSIX ABI "LAMP stack," using split drivers for sensors and delegated sockets for networking. 43
9 KaTZe benefits from the same application optimizations that Linux does for SQLite3's speedtest1 benchmark. 44
10 KaTZe's socket implementation provides reasonable networking performance both in connection latency and throughput with full delegation to Linux. 45
11 In our proposed architecture sensor data is securely sent to the trusted application. 49
12 Re-using drivers can be achieved with the "sledgehammer" approach that hosts a complete VM, here denoted "DD/OS". Figure from [51]. 52
13 Block model of a modern ISP, the ARM Mali-C55, showing considerable internal complexity. Image from ARM [8] 53
14 The logical structure of the on-board ISP is several sub-devices. 59
15 Our driver model configures the ISP with a userspace library which is effected by kernel drivers to hardware, without device abstraction. 61

16 SOD achieves a recognition speed of 27 frames per second on KaTZe compared to Linux's 126. 64
17 Decomposing a device reveals some have heterogeneous security properties which permit secure shunting of functionality out of the TEE. 67
18 Our proposed driver model spans both the untrusted and trusted OS, delegating platform management to the untrusted partition. 70
19 The functionality of a single "composite" paravirtualized device is provided by several discrete hardware devices. 71
20 Our paravirtual framework organizes the system's devices into three perspectives, or "layers" of increasing granularity or abstractness. 72
21 Paravirtual drivers expose a "key" to the untrusted partition, which various underlying platforms can fit to provide services to the TEE. 73
22 If input A linked to output B are trusted and a midpoint C cannot be configured to output to an adversary, then C's *data* is trusted even when *control* is not. . . 75
23 By changing only the selected Linux driver, our implementation re-uses valid Linux configurations to enable a trusted paravirtual device. 76
24 The HTU21D paravirtualized driver achieves a reasonable fraction, 0.626x the performance of a native driver equivalent. 79

Preface

I recall in the first year of my graduate studies how others explained the process loftily as "you will expand the frontiers of human knowledge," with emphasis on *you*. Now that I am at the end of that process I understand that this could not be further from the truth, and in fact I owe an incredible debt to so many around me, without whom I could not have done this. Time is the only resource we can't get back, and these people have given me so much of their most precious resource. This is to thank them for their generosity and make sure they get their deserved share in this achievement.

My friends and family: to my partner Sam, for all she has done. Her love and support is immeasurable and I could fill pages and it would still not be enough. To my grandmother, always Nana to me, for her unconditional support and encouragement and always, always believing that I had it in me to do this. To my late grandfather, Papa, who gave me a love of computers and technology that survives to this day. To Buck, who reminds me that anything of value requires sacrifice and struggle to achieve. To Izzy, who reminds me that we all face difficult struggles, but it's easier if we face them together. To the rest of my family, my mother, my sister, and Carole who remind me that far does not mean forgotten. To the many friends I have made in Pittsburgh: Jenn, Matt, Emily, Maddy, Matt, Isabella, Savannah, Ethan, and the many others for the social fabric they so seamlessly wove me into.

My academic friends and colleagues: to my advisor, Jack, for his principled mentorship, for his high standards that continually pushed me to learn, grow, reflect, complain, and finally to understand. For teaching me to always strive to do the best you can and that it will not be easy but it will be worth doing. For his concern and investment of time and effort in my growth as a researcher. To my other committee members for their unique perspectives and for tolerating an inconvenient timeline; their input has substantially improved the content and course of this work. To the `#osdev` community who were a sounding board for many research ideas and who taught me a great deal along the way. To those who read this manuscript and suggested improvements for their perspective, insight, and even more of their time.

1.0 Introduction

Trusted computing has become a key computing domain and has joined the ranks of primary design goals in many use-cases. As computing becomes more ubiquitous with the continued expansion of IoT, edge, and mobile computing, data privacy concerns mount. The dominant model for trusted computing is the "enclave" model, wherein trusted services are provided by a codebase that's minimized to improve safety and limit complexity. However, as the kinds of workloads deployed to trusted execution environments (TEEs) increases from things like secure key management [62, 63, 34], mediating access to random number generation [4], or DRM-enforcement [27] to complete data processing pipelines, the enclave model falls short in a number of ways. Examples of these data pipelines can be seen in the contexts of "smart shopping" experiences where cameras monitor shoppers and track what they purchase [37] to facial recognition deployed in public spaces for tracking the spread of infectious diseases [69]. These use cases promise to dramatically expand the amount of information gathered and so will have large privacy implications for the broader public. In these systems, data confidentiality is of great concern, not just for the system owners but also the subjects of the collected data. Supporting the deployment of these data pipelines into trusted infrastructure with appropriate security safeguards and privacy protections will be a critical need in the near future. Fully supporting this use case requires an end-to-end TEE architecture that incorporates secure I/O, data processing, data storage, and data transfer. In addition to providing effective security compartmentalization we would like to move data processing as close to the sensors as possible in order to minimize the attack vectors that come from data movement and distribution. This creates a need for IoT devices in question to support secure access to local sensor devices as well as the ability to support comprehensive data processing pipelines locally on the IoT platform.

An example of such a trusted sensing environment is shown in Figure 1. In this system a collection of IoT devices is deployed into a physical space with various sensors to collect information about the environment. The collected data will include multimodal information aggregated from multiple types of sensors and capture the state of the physical space as well

as any individuals inhabiting that space. We assume that this information will be directly beneficial to the owner/operator of the infrastructure and will also potentially provide benefits to the individuals inhabiting that space. At the same time, the sheer scale of data collected from the environment will pose significant privacy risks for the individuals in the space, and so will require some level of privacy safeguards to be put in place. In our model, we propose a technical solution to enforcing privacy safeguards. We assume that infrastructure owners/operators will be compelled, potentially via legal mechanisms, to perform data collection, aggregation, processing, and storage inside an environment that provides privacy protections in a way that is verifiable to the individuals whose data is being collected. The foundation of our approach is hardware-based Trusted Execution Environments (TEE), and their ability to provide trusted, secure, and attestable hardware partitions to isolate sensitive operations.

TEEs allow a single hardware platform to be partitioned into trusted and untrusted domains. The trusted domain ensures that any code being executed is tamper-proof and that any data being processed by that code is likewise secured from the untrusted partition. In addition, TEEs generally also provide mechanisms to remotely attest both the integrity of the trusted domain as well as the authenticity of the code and data contained in it. By supporting both untrusted and trusted domains on a single platform, it is possible to deploy a distributed architecture that allows an operator to directly configure and manage the infrastructure to support their needs, while restricting what they are allowed to do with the

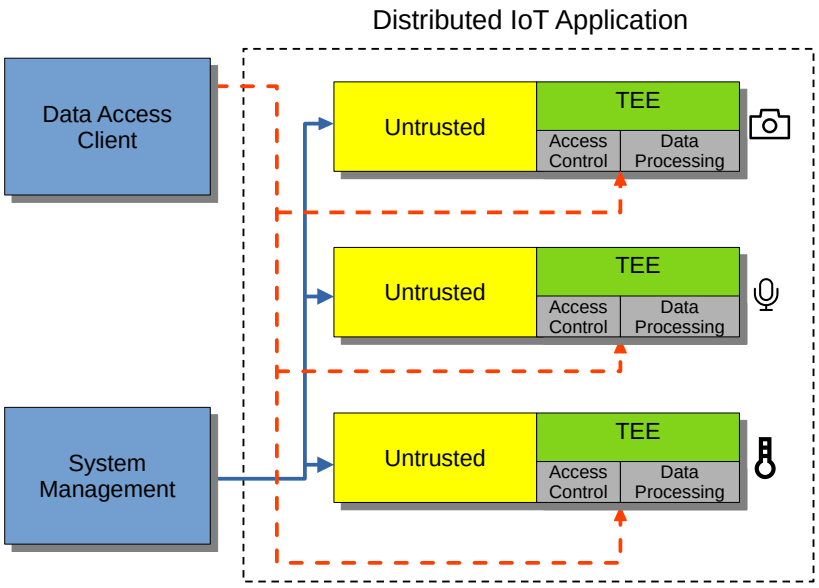

Figure 1: Processing data at the sensor and separated management and data access planes are key components of our target class of applications.

data the infrastructure collects. With this arrangement the infrastructure can be separated into an untrusted control plane and a trusted data plane, thus enabling secure and privacy-enforcing data collection in an otherwise untrusted environment. Our goal is to balance the requirements of both the owners/operators of the infrastructure as well as the individuals whose data the infrastructure is collecting.

Currently, we do not know of a system that provides this capability: using unmodified applications built for existing, typical deployments such as Linux and other UNIX-based systems, while exposing to those applications secure I/O to sensors, all in the face of an adversary like the system owner/operator who can deploy arbitrary, privileged software to the untrusted system partition. The deficiencies of existing systems falls into three categories: the TEE OS architecture and the applications it supports, the direct I/O capability, and the security confidence in such a system, generally in terms of the trusted computing base (TCB).

In modern TEE architectures, enclaves are expected to provide services with limited scope [99, 4], and the trusted OS kernel is limited regarding resource ownership and allocation, device ownership, and the kinds of application requirements it is designed to support in contrast to the comparatively much richer application environment found in the untrusted partition, where the majority of application workloads currently reside. However, as new classes of secure IoT sensing applications evolve it will challenge this "trusted service" model as more primary application workloads move into the trusted domain. Further, in the current enclave model applications must be built with a hard trust boundary in mind, forcing programmers to carefully consider the data that crosses the boundary [26]. In addition, the application APIs/ABIs exposed by modern TEE environments do not support modern classes of applications, and often require extensive engineering and design work to adapt existing solutions to a TEE architecture [68, 113, 67, 54]. These efforts introduce both additional development hurdles as well as the possibility for security-compromising bugs, thanks both to increased complexity and the inability to use already hardened and tested software packages [107].

Put simply, existing TEE OSes are not designed to host applications that are both general purpose *and* secure, or to provide portable abstractions for direct access to secure I/O. To

address this issue we propose the use of lightweight kernels [85] (LWKs) as the trusted operating system in order to support a trusted execution environment that directly supports a wide range of modern applications. The lightweight kernel is an OS architecture that emphasizes design simplicity while implementing enough interfaces and features to provide a familiar environment that is widely portable to existing applications. LWKs by design do not provide all of the features and capabilities of "full-weight" kernels (FWKs) such as Linux or FreeBSD, but rather identify a baseline of functionality that provides widespread portability in as simple a design as possible. The end result is an application runtime environment that provides many of the same basic process, file system, I/O, and memory abstractions as a FWK, while not supporting the entirety of the FWK's ABI. This allows application binaries that leverage common and standard ABI features to run directly on an LWK without modification and often without re-compilation. While LWKs will obviously not support the full range of applications that a FWK does, we claim that its compatibility is sufficient to support the classes of applications likely to be used on an IoT platform. Ultimately, we are faced with a balancing act between maximizing application compatibility and minimizing the attack surface of the TEE, and we believe LWKs occupy an attractive middle ground in this continuum.

However, the OS and applications form only one part of the trusted sensing system. These platforms also possess diverse sensors ranging from the simple motion detecting sensors to the very complex, such as cameras that are themselves composed of semi-independent components on the same module. One of the primary advantages in this regard that FWK kernels possess is their robust driver support, driven both by manufacturers and independent maintainers/developers. Although this situation suggests a viable path for secure sensing could lie in hardening a FWK, approaching the problem from the "opposite end" so to speak, the complexity inside FWKs in terms of drivers and kernel infrastructure needed to support diverse device hardware architectures makes analysis and security confidence extremely difficult. Existing theoretical work to prove OS correctness has been limited to comparatively simple microkernels [40] which moreover require specific behavior from the hardware, reducing their generality. Further complicating this idea is that security exploits are regularly discovered in mainstream production kernels like Linux [39], a problem

essentially guaranteed by the continual evolution of the kernel in different, simultaneous directions.

Instead, directly driving sensors in the trusted partition is feasible, which we support with two observations: first, the enormous diversity of devices supported by mainstream kernels like Linux necessitates a similarly complex driver and kernel infrastructure. By rather specifically targeting trusted sensing, we make irrelevant the majority of the cumbersome driver infrastructure that mainstream kernels contain in order to support a wide variety of devices. We claim that in stripping back these layers of complexity it is revealed that the drivers for relevant sensors are in fact quite self-contained. The second observation is the aforementioned split between the *control* and *data* planes. By this we mean that without introducing data leakage it is possible to delegate control of a sensing device to the existing FWK where the device driver is already supported, but move the dataplane portion of code into the TEE itself. This split is similar to the model found in common virtual drivers, such as virtio. In this context, VMs act as clients and treat the underlying hypervisor as a service provider, requesting access to virtualized, generic devices, requiring the hypervisor to translate or otherwise coordinate virtualized access to physical devices. This "generic device" model hides hardware complexity from the client. However, our model is different both in that the device data must be hidden from the service provider which controls the device, as well as that our model arranges resource partitions *horizontally* across the trust barrier as opposed to *vertically* as is the case with hypervisors.

1.1 Motivation: Autonomous/Smart Shopping

Distributed applications with privacy as a core motivation are problems we presently face. Recently retail stores have introduced shopping experiences where customers don't have to interact with any employees to purchase items and leave the store, perhaps most prominent is Amazon's "Just walk out" or "Amazon Go" stores [37]. In these stores sensors monitor customers as they shop, identifying their picked items as they go, and finally customers can seamlessly leave the store without having to present their items at a checkout. This

is accomplished with sensors across many modalities widely ranging in complexity from computer vision that performs person localization within the store to weight sensors on shelves to help identify when items are taken. This rich, multi-modal information is used to deliver the employee-free shopping experience, but the lifecycle and handling of that data is almost fully in the control of the venue owner. This position creates a binary problem where in order to get the shopping experience a customer must surrender control over highly-identifying personal data or decline the shopping experience altogether. This problem exists because there is a lack of technical solutions to control and regulate how a venue owner uses a customer's data. With such a solution, customers could define a data use policy that venue owners must adhere to and in exchange venue owners can, for instance, incentivize customers or "buy" their data by offering augmented experiences. The most permissive data use policy may earn the full autonomous experience while the store owner may decide that a more restrictive policy may only be rewarded with coupons or similar–what matters here is the system that enables this capability. Additionally to the economic argument we expect venue owners to face increased legal pressure to provide such a system, such as from the EU's GDPR [76]. In the case of legal compulsion a technical solution also provides value to the venue owner as evidence of compliance with the legal privacy framework.

The Privacy Backplane research project tries to address the autonomous shopping use case[46], and the requirements of the backplane project have greatly influenced our designs. Further, the Privacy Backplane is a concrete example of the application trends introduced above and illustrates the complexities in designing rich, trusted system software. A "privacy backplane" is a distributed network of low-cost nodes, such as IoT-oriented single-board computers, which securely gather and process privacy-sensitive data. Importantly, the node and infrastructure owners may be untrusted, requiring the partitioning of each node in trusted and untrusted partitions. The untrusted partition is used in the ordinary way for edge/IoT contexts: load balancing, management, deployment of services, etc. The trusted partition owns all sensors and securely collects and processes data from those sensors, which range from simple motion detectors to complex cameras. The collected data contains sensitive information about *third parties*, that is, entities other than the infrastructure owners. These entities supply to the backplane a policy that determines what kind of actions on their data

are allowable. Each backplane will define its own "privacy types" that user policies interact with. To make this collected data useful to the infrastructure owners, the backplane will support the deployment of operators which transform collected data in some arbitrary way, specified by privacy types. Examples include anonymizing operators, such as object identification computer vision applications; the data contains enough information to identify the entity at the input, but the output of this operator *cannot* identify an individual. In designing a TEE that supports a Privacy Backplane we face several key problems or requirements which we consider in the course of this work:

1. An OS environment that supports "arbitrary operators" as well as software needed to run such a distributed application.
2. Diverse I/O capabilities, at least including first-class networking and secure access to sensors at potentially every node.
3. Data processing as close to the sensors as possible to minimize security risks.

These problems will be important design points for us to consider later on.

1.2 Thesis

We assert that lightweight kernels (LWK) are an ideal foundation to build a TEE that supports general-purpose application workflows and provides direct access to secure I/O peripherals even in the face of a privileged, adversary. By striking a balance between simplicity and usability, LWKs provide a familiar application ABI with a slim TCB, expose a rich OS environment, and allow applications to more directly leverage hardware. We describe and implement such a system for the ARM TrustZone platform, including a kernel, an I/O stack to securely drive sensors including cameras, and develop a virtualized driver model to allow secure re-use of existing device drivers.

1.3 Research Contributions and Insights

This thesis is organized into discussions of the following research contributions, detailing the design, implementation, and evaluation that lead to the following insights:

1. Multi-kernel computing architectures are ideal to address the problems TEEs face. Instead of the isolated "secure enclave" notion, the trusted partition should be a first-class partition with differing hardware and software requirements to the untrusted partition. The trusted partition should *not* be fully minimized because this leads to inflexible solutions that cannot easily grow and adapt to changing TEE workloads. We discuss this in Chapter 3.
2. TEE hardware has improved in flexibility and so full-featured OSes capable of handling diverse workloads are feasible and practical inside modern TEEs. Future TEE OS designers should reconsider the use of highly-specialized OSes where possible and recognize that sophisticated hardware allows rich OSes while preserving security. We discuss this in Chapter 3.
3. System partitioning at the hardware level must be extended to all parts of hardware. Existing, older TEE architectures that only allow the TEE to use a subset of hardware are not adequate for modern TEE workloads. This has been recognized by current and upcoming TEE architectures. We discuss this in Chapter 4.
4. Tightly paired with multi-kernel computing architectures, the trusted partition software should *leverage* the untrusted partition whenever possible. Multi-kernel systems encourage OS composition which can be used to enable powerful functionality cheaply in the trusted partition. We discuss this in Chapters 4 and 5.

A brief discussion of the contributions and their significance follows.

1.3.1 Trusted Execution Environment (TEE)

The foundational contribution provides a trusted operating system and runtime, specifically one that supports existing applications currently in use in IoT environments. Currently, IoT deployments are becoming more and more pervasive and so the "cumulative mass" of

applications deployed grows. Security concerns are increasing in tandem and the greater the inertia of these IoT deployments, the more painful it is to re-engineer them to target existing TEE solutions that impose a strict and cumbersome application framework. By providing an alternative runtime based on lightweight kernels, which we call KaTZe, which exposes the familiar UNIX-like application programming environment *inside* the trusted envelope, existing application deployments can be re-used with little to no source modification; most applications can be targeted to our runtime entirely at the build configuration.

This provides an insight: a full-featured, mainstream kernel like Linux is not required to support relevant application workloads. Instead, specializing the kernel allows us to simplify the TCB, providing qualitative security improvements by eliminating the vast majority of code that is needed to support the many edge-cases that make up "general-purpose" use.

1.3.2 Secure Peripheral Device I/O

As trusted computing expands in the mobile and IoT space it is acquiring new responsibilities in addition to the traditional secrets management and integrity verification. Increasingly important are distributed trusted applications and trusted sensors, both of which require secure access to I/O. However, existing kernels like Linux achieve this with many hardware abstraction layers (HAL), both in the kernel and in userspace. Our work enables secure application I/O with substantially less complexity, partly by more directly exposing hardware to userspace. We show that it is possible to support many devices, including those as complex as cameras, largely by re-using existing drivers with their HAL parts removed.

The insight: much of the complexity present in mainstream kernels' device models arises from being generalist kernels that seek to support *all devices* on *all hardware platforms*. In targeting a specific class of applications on a specific type of hardware platform we can trim much of this fat without losing functionality.

1.3.3 Driver Paravirtualization

We observe that the complexity in existing drivers is of two kinds: core device functionality and integration with the broader kernel systems. Additionally, core device functionality

can be classified according to its security sensitivity. We recognize that a device driver has *control* and *data* components and that only the data components are security critical. We describe this device model and use this model to build a paravirtualization framework that splits drivers across this axis. We show that in the context of shared, untrusted infrastructure this poses no greater security risk than if the TEE contained *both* the data and control components.

The final insight is that as trusted application deployments advance, acquiring services from an untrusted kernel becomes a significant paradigm, as opposed to the current situation where TEEs provide limited, secure services to untrusted applications. This paradigm mirrors the current one by eliminating unneeded complexity in the TEE, but achieves it by delegating to the untrusted kernel instead of being delegated to. We gain another advantage from paravirtualization in terms of portability. By hiding control behind the paravirtual curtain on the untrusted side, that untrusted side can potentially change without requiring changes in the TEE itself. This allows for a TEE image to be somewhat hardware-agnostic yet still secure.

2.0 Threat Model

Underpinning any security-focused system must be the threat model. As a primary use case, IoT/edge platforms motivate our threat model discussion, but we do not limit ourselves to that platform. In our model we have several actors:

- The *untrusted* infrastructure owner, who deploys and manages the system nodes. This owner can install arbitrary privileged software to the unsecure partition.
- The *trusted* secure partition, in particular the software stack spanning from the secure bootloader to the secure operating system and applications.
- The *trusted* third party, who "provisions" system nodes with a root of trust in an immutable way before the infrastructure owner can deploy system nodes for secure use. They act as a trust anchor.
- The owners of collected data. They are primarily stakeholders in the system and have an interest that their data is used consistently with their policies.

In particular, the secure software stack must be agreed upon by the trust anchor and infrastructure owner jointly. This requirement hinders the infrastructure owner from deploying trivially-backdoored software into the secure partition as part of the requested deployment. Once a system software image is assembled by the owner it must be audited by the trust anchor before cryptographic signing. The trust anchor must be publicly known so that data owners can decide whether to trust that a specific system has been audited sufficiently. The general architecture of our threat model is shown in Figure 2.

With this system, we make the following guarantees:

1. The infrastructure owner cannot acquire data from owners that has been collected by sensors without the involvement of the secure partition. That is, the secure partition always moderates data release to an untrusted actor.
2. The infrastructure owner can see that secure partition devices and sensors are active.
3. The secure partition software has an immutable cryptographic signature. A new system software image can be deployed, but this image has a new, distinct signature for the

purposes of secure applications. In other words, the infrastructure owner cannot modify the secure partition software without requiring reverification of the software stack.

This system provides these guarantees under assumptions concerning mostly physical attacks and hardware implementation. Physical attacks are not considered, as tamper-proof devices is a well-developed industrial concern and protecting against physical attacks is largely orthogonal to protecting against attacks in software. This means that we do not consider attacks like direct bus snooping or chip delidding. Further, the platforms that we deploy to will be controlled by a potential adversary and we cannot trust the entire hardware platform. If we could, existing security techniques would be applicable and this work would not be necessary. While a main focus is to secure sensors to prevent misuse of captured data, it is not generally possible to prevent the device owner from installing their own, untrusted sensors in parallel to trusted ones. Without a wider, more intrusive security technique, such as controlling all system busses and exposing a virtual interface to the untrusted partition, we cannot prevent, for example, a separate camera being connected and made available to the untrusted partition to effectively capture the same data as the existing, secure camera. Thus, we require that, once deployed, a system's hardware configuration does not change–whether this is effected through tamper-resistance techniques such as potting or due to legal enforcement does not affect our work.

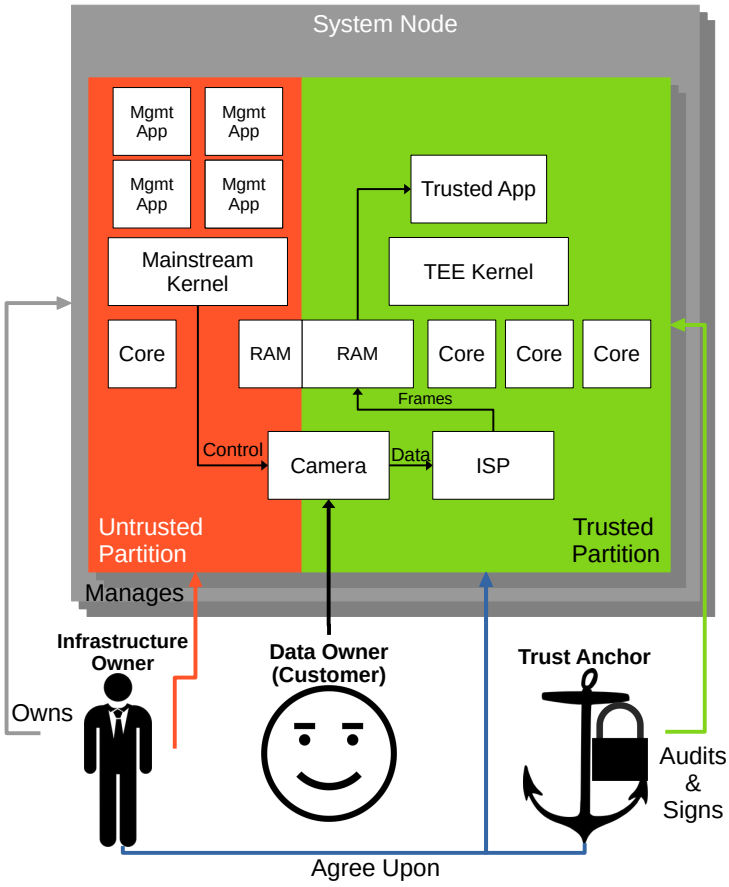

Figure 2: Our threat model ensures that secure partition software cannot be tampered with by the owner.

We assume that any security hardware is correctly implemented according to its specifications. We do not consider side-channels, since they can be used to defeat compliant hardware anyway [14, 53, 41]. We assume that any deployed systems are provisioned securely, essentially important for a secure boot scenario. As part of the secure boot process, trusted keys must be present and permanently bound to the device to serve as a root-of-trust which each stage in the boot process can refer to for verification. This assumption provides us Guarantee 3 in the threat model and is a commonplace measure taken in secure systems currently.

A primary use case of this work is distributed, IoT sensing environments. Accordingly systems may be in public locations without complete physical security, and a large variety of software may be run for administration purposes in the untrusted partition. We have two types of adversaries: infrastructure owners who can install arbitrary, privileged software into the untrusted partition for the purposes of managing the infrastructure in an honest but curious manner [1], and non-owner adversaries whose intention is to subvert or compromise system behavior. Since infrastructure management responsibilities are offloaded from the trusted environment, we assume that the infrastructure operator will ensure that system availability is maintained and that it is in their direct interest to do so.

Finally, we do not explicitly attempt to support mutually-distrusting applications in the secure partition, instead assuming that the system owner will vet deployed software and exclude malicious software. This assumption is supported by two facts: that attestation, briefly described below, requires the entire secure system software image to be signed by the trust anchor. This implies that any signed software image has been audited by at least two agents, both interested in the continued good operation of the underlying system. Further, changing the underlying privileged software will be generally unnecessary and even changing the userspace software image will be uncommon. Instead, we envision that software sandboxes such as WebAssembly runtimes will be deployed, providing a reasonably-secure vector for less-trusted software to be deployed into. While traditional, existing operating system design provides isolation mechanisms from trusted applications misbehaving, we contend that given the above factors, we do not consider that one trusted application may be adversarial to another.

2.1 Attestation

An important component of a secure system's functionality is attestation, both remote and local. These attestation types are especially important when developing a trusted distributed system, as without any form of attestation it is generally impossible to establish the trustworthiness of a communication partner. Local attestation, referring to the capability through which running software knows the state of the hardware it is running on, typically since the last reboot of the system. This provides software with some assurances about the possible configurations of the hardware as well as the other software running on the system. Remote attestation allows an inquirer to learn something about the state of hardware and software of the attester. Both types of attestation require the involvement of a trusted third party, sometimes called a "trust anchor." A more thorough discussion of this system is given in Section 6.1, but the important point for this section is that our "trust anchor" from above creates a binding between a cryptographic identity and the specific hardware system, stored privately. This identity is immutably bound to the hardware, providing a trusted reference point for attestation. This trust anchor must not leak the private key for the system, as once that private key has leaked, the hardware this key is bound to is never again trustworthy. Mitigation schemes for this vulnerability exist, such as the use of physical unclonable functions (PUFs) that mean that impersonation using only the private key is not possible [97].

3.0 LWKs as Secure OSes

In our first chapter we aim to show that the lightweight kernel model possesses ideal properties and capabilities to implement a TEE OS that supports general-purpose applications and the relevant TEE hardware, including laying the groundwork for complex sensors and delegated drivers, discussed in the later chapters. We lay out this design and then describe our implementation of this design. We identify typical applications for the secure, edge/IoT context, run them without modifications, and demonstrate good performance. We do this with similar complexity to existing TEE solutions, demonstrating that the existing shortcomings are due to differing design decisions, not infeasibility. Focusing on the architectural compatibility between TEE hardware and a full-stack OS environment, we conduct micro-architectural benchmarks as well as a fully-composed application to demonstrate generality. We use a typical "LAMP" stack that's directly relevant to the edge/IoT computing context. But first, we provide a background on the most important concepts and describe the state of the art along the way.

3.1 Background and Related Work

3.1.1 Trusted Execution Environment Models

At the bottom of any "trusted system" is the term "trusted execution environment (TEE)," which describes the properties of the system and not a particular implementation, in the same way that "mobile" or "cloud" computing does. Different hardware platforms offer widely varying properties to leverage in the implementation of a TEE, and when discussing computing security and TEEs, the "CIA triad" is used, first coined perhaps in the 1970s [70]:

1. *Confidentiality* – Unauthorized parties cannot read trusted data.
2. *Integrity* – Unauthorized parties cannot modify trusted data.
3. *Availability* – Trusted data is available for use by authorized parties on-demand.

TEEs provide some combination of these properties through a combination of hardware and software mechanisms. For example, Intel's SGX platform provides stronger guarantees by providing in-place memory encryption and integrity checks with machine attestation as a hardware feature. [35] On the other hand, implementations like ARM's TrustZone provide comparatively few properties, offering only access control mechanisms at the hardware level, leaving TEE implementations to provide other security properties in software [90]. However, in exchange for stronger security properties, TEEs like Intel SGX must endure hardware-based limitations, such as a strict memory limit in the low hundreds of megabytes–though later revisions of SGX in newer hardware remove this limitation at the cost of losing in-place memory encryption. By contrast, TEE implementations with less advanced, integrated hardware features like TrustZone, various academic RISC-V TEE designs [23, 20, 109, 50, 10], or OpenPOWER's PEF [34] enjoy increased software flexibility and aim to provide missing components of the triad in software.

A key property of a TEE is its *root of trust*, which refers to the model in which software comes to be trusted. In most TEE platforms today [90], the root of trust is static, typically realized by writing an immutable cryptographic key into the hardware at manufacturing time. Then, through *trusted boot*, earlier boot software can verify that subsequent software is trusted if it is signed by this immutable key. Some systems like Intel SGX and AMD SEV provide a dynamic root of trust wherein trusted software can be launched from untrusted software. These schemes are more complex and how this is achieved is beyond the purposes of our discussion. Overall, this scheme provides some guarantees of correctness in that a manufacturer will be strict about the software that it signs–Intel is unlikely to stake their reputation and sign unfamiliar, unaudited code.

3.1.2 System Software Assumptions and Application APIs

TEEs can also be classified based on the architecture of the software, broadly across two axes: whether software deployed to the TEE can be third-party, and what capabilities and interfaces the TEE offers those applications. The first axis, third party application support, denotes whether a given TEE allows software other than that included with the

TEE to be used. While first-party-only software encourages coherent software design and fewer potential attack vectors for insecure or compromised software it obviously restricts the application set to that which the TEE developer is interested in. Current TEEs, such as Android Trusty [4], follow this model, with all software deployed to the TEE being part of the same source tree as the Trusty TEE itself. More general is OP-TEE [99], where arbitrary applications can be deployed and run in OP-TEE, but only a limited featureset is exposed by the kernel, and applications must be constructed around a specific API. The ends of this axis can be broadly described as "specialized" or "general-purpose" APIs. To be clear, when we say general-purpose APIs we mean those that are in mainstream use today for ordinary programs: UNIX-like/POSIX-like syscall interfaces and runtime environments, with access to ordinary OS abstractions like multiprocessing, filesystems, and network sockets. Thus we mean by "specialized" APIs those that discard some number of these "typical" features. Most, if not all, current mainstream TEEs require the use of somewhat specialized APIs, ranging from the GlobalPlatform TEE API which requires applications to have secure and non-secure halves that explicitly marshal data across the trust boundary using a message model, to the minimally-intrusive APIs of Android Trusty that provide a mostly-ordinary C application runtime that exposes a different set of syscalls than one expects from a UNIX-like OS.

However, utilizing these existing TEE APIs is cumbersome, since applications of some complexity will need to request services from the FWK service kernel, which requires sending messages across the trust boundary, which is exposed to userspace applications directly. Existing applications must be designed around this API, resulting in limited or no immediate code re-use.

This model has a strong assumption that *only* the security-critical parts of an application's logic will be hosted in the secure enclave, done to strengthen security considerations and reduce the likelihood of breaches, accidental or otherwise. This model is fundamentally incompatible with the use-cases we identified, as applications grow in complexity, more kinds of data processing will be in the secure system partition. Thus, accommodating these complex application systems requires re-engineering according to the existing TEE APIs, or rejecting those APIs, which we do. We contend that such re-engineering is plainly in-

feasible, and instead a more general-purpose TEE OS should follow the lead of existing general-purpose OSes and provide a familiar program environment. However, we want to avoid the increased complexity that modern FWKs such as Linux have accumulated over the years, and so we target the "common case" of broadly POSIX-compatible APIs, eschewing notorious Linuxisms that some applications have grown to rely on for improved performance, such as epoll. Instead, we prefer portability where possible, roughly following the example of the BSD OSes.

To be clear, our use case of driving applications workloads from the trusted world means we are inverting priorities, retargeting the untrusted OS as the "service OS" instead. While we may interact with the untrusted world, we anticipate large portions of software systems to have lifecycles fully inside the trusted space, with most untrusted world communication taking the form of responsibility offloading for security non-critical paths, like network stacks when the transport-layer security protocol (TLS) is in use. This difference in model wants a full-featured, familiar programming environment for applications such as typical libc support and even high-level language support such as WebAssembly or Java. As part of the existing application programming environment, support for a POSIX-like system call use model is also important.

As trusted applications become more complex and aim towards general-purpose uses instead of security-critical functions like key management or randomness sources, the diversity of needed resources also increases. A straightforward application runtime like Intel SGX's SDK provides a way to develop trusted CPU operations and encrypted memory but does not provide networking, filesystem access, etc. This is offloaded to the untrusted caller, requiring the trusted application to encrypt data. As stated previously, to extend trust into the I/O plane a runtime hosted by an untrusted OS cannot access this peripheral hardware directly, and extending the runtime to integrate with the untrusted OS comprises a set of features that's broadly equivalent to an OS. That is, implementing support in such a runtime, we claim, is similar to implementing a trusted OS that's tightly coupled with the untrusted OS. In other words, the spectrum of application runtime environment stretches from runtimes that support only classical, CPU-based applications, like WebAssembly, to a complete OS that includes libraries for interacting with the entire system. We contend that the closer to

the OS end of the spectrum you get, the harder it is to *not* realize the application runtime as a general-purpose OS, and trusted applications are continually pushing in this featureful direction.

3.1.2.1 Microkernels and Unikernels

An important design criterion is minimal or no re-engineering of existing applications. As such, any operating system must provide or emulate the full range of typical OS abstractions that mainstream applications expect. Most existing microkernels provide some sort of Linux/UNIX compatibility layer, but do so at a performance cost. While the use of a microkernel is possible [38, 40], we contend that a microkernel with a well-integrated compatibility layer necessary to support these applications would be better served by more tightly and efficiently integrating this compatibility layer into itself. In other words, if the target workloads are broadly POSIX-like, then the underlying kernel does not benefit as much from the scalability, composability, and design cleanliness that microkernels typically offer.

Unikernel systems such as Hermitcore [48], Lupine [43], UniGard [91], and others are interesting alternatives. Unikernels present their own problems–most notable here is how to multiplex underlying hardware. TrustZone does not have a strong notion of inter-enclave separation, and so the use of unikernels to implement trusted multiprocessing mostly offers improved assumptions of correctness and crash resilience, as a malicious or misbehaving unikernel cannot be prevented from corrupting another's memory. Additionally, implementing multiprocessing requires a robust, flexible inter-unikernel communication method, which again either requires the coordination of a hypervisor or manual coordination of shared memory channels. We conclude that the logical correctness and application encapsulation benefits offered by unikernels are largely outweighed by the requirement of a hypervisor for separation.

3.1.3 TEE OSes and Software Runtimes

Highly related to the application runtime that TEEs expose to the applications is the abstractions and features that the underlying OS provides. We previously described *how* these features are exposed to the application, we now describe *what* features are exposed. TEEs again can be placed on an axis from minimalist to maximalist. Further, the featureset that the TEE exposes indicates what kinds of software the TEE designers thought might be used. Systems like SCONE [9] lack a notion of a trusted OS altogether and provide only an application runtime, relying on underlying hardware security to protect trusted code, whereas systems like TrustZone's OP-TEE [99] implements a complete OS environment with a particular application SDK that applications must target. Shinde's Panoply system [94], which provides a typical OS abstraction environment within Intel SGX enclaves, also does not address the related problem of I/O ownership. In almost all of these systems, however, is the implicit notion that trusted software provides a specific, limited service that untrusted software uses, which we call "service OSes."

Existing solutions for TrustZone TEE OSes generally fall short by either being fundamentally unsecurable or by not supporting the target class of applications, whether due to an incompatible application environment, insufficient resource allocations, or a lack of a secure I/O pathway. Full-weight kernels like Linux or BSD-likes are full-featured at the cost of complexity, which balloons the TCB. Linux is at the extreme end of complexity and features, which introduces a very large surface area of code which may contain serious bugs, with privilege escalation attacks being not uncommon in Linux and regularly discovered. In untrusted computing contexts this concern can be mitigated with resiliency and fault isolation techniques that prevent a system crash in the event of software bugs. Trusted applications, on the other hand, are designed with heightened expectations for correctness and security to protect sensitive data. Linux and similar kernels' high TCB size thus make them generally unsuitable for use as a trusted OS.

3.1.3.1 Runtime Systems

Due to the expectation of running ordinary, POSIX-compatible workloads that take advantage of a POSIX-like OS's application environment and abstractions, existing runtime-based systems like Tarnhelm [83] and TLR [89] that provide trusted computing by various means using runtimes are not suitable. TLR is an early example of a language runtime implemented for the TrustZone platform. The TLR system extends the .NET runtime with primitives that allow specific portions of code to be run inside a TrustZone-protected enclave. As a managed language runtime, TLR lacks most of the abstractions that modern OSes offer, and they do not attempt a generic pathway for secure I/O. Tarnhelm is a more recent work that allows sections of code to be run in a TrustZone enclave with compiler annotations, allowing single-source applications to span the trust boundary transparently, at the function call-level. However, their underlying TrustZone OS is based on OP-TEE and inherits most of its deficiencies.

3.1.3.2 Service-like OSes

The OS's capabilities inform the expected uses of the software; an OS without preemption requires applications to be carefully constructed to voluntarily cede control back to the kernel, or else the system has no multitasking. The majority of TEE OSes we have deemed "service OSes" because the features they expose indicate that applications deployed to the TEE should be just that, services. Indeed, many adhere to a standard known as the GlobalPlatform TEE Client specification [26], and their design indicates clearly that untrusted applications are the primary concern on a trusted computing-capable system. Many existing TrustZone TEE OSes such as OP-TEE [99], Android Trusty [4], and the seL4-based Micro-TEE [38], despite being different underlying kernel architectures, all deliberately target this RPC-based API and limit the class of target applications to "trustlets" that provide trusted services. GlobalPlatform is an industrial standards organization that creates vendor-neutral technology and specifications for trusted computing systems. This standard, endorsed explicitly by ARM with their reference firmware implementation, TF-A [100], imposes this service-like model onto applications. This dominant standard has resulted in a magnetic

pull, and much current TEE research is predicated on the use of the GlobalPlatform standard. Consequently, adapting these OSes to support general-purpose computing requires substantial redesign and re-implementation.

By adhering to this architecture, existing TEE OSes have limited their own features and reduced the complexity of applications they can support, resulting in a poor match given current trends in trusted applications. Currently, TEE OSes are treated as second-class citizens on the system and are given limited resource ownership. Often TEEs have small memory allocations and no direct ownership of hardware resources, not even the CPU cores they run on. For even embedded AI models with memory footprints in the gigabytes that struggle to run on Linux on these systems already, these intentional resource limitations preclude this class of application in the current trusted space. Our use case requires that applications running in the trusted space be considered the *primary* component of the system, rather than a service component, which rules out this dominant "service OS" paradigm.

Haven [12] and Graphene [18] allow unmodified applications to run in SGX by inserting library OSes underneath the applications. Haven achieves this with the Windows ABI by refactoring the Windows kernel into a user-mode library OS, eliminating the need for a privilege mode switch. Graphene achieves this on Linux with the use of a library OS layer that sits underneath enclaved applications, intercepting and translating system calls. Haven and Graphene both explicitly argue our point that a full-featured OS is unsuited to the SGX platform, remarking that putting OS semantics into an SGX enclave requires emulating several common OS features. These works also do not address the important I/O problem. Shen's Occlum system [92] conceptually extends the Haven system by providing inter-task isolation under SGX.

OP-TEE [99] is currently the most widely used TEE OS for TrustZone, enjoying mainline ecosystem support in Linux, the ARM trusted firmware, and the common U-Boot bootloader [103]. OP-TEE closely follows the GlobalPlatform Client TEE specification, though not quite implementing all of the API, leaving out things like TLS connection termination into a trusted application [72]. OP-TEE runs in a small amount of RAM and services requests by context switching on a core and running small trusted applications with limited resources. The OP-TEE OS core runs in trusted space, while a supplicant program in Linux facilitates

sending messages to OS core. OP-TEE presents OS-like trusted services as "pseudo-trusted applications" that a client application must program against and call, not entirely unlike the microkernel userland service model. Currently, the official OP-TEE distribution provides drivers only for security-related platform hardware such as clocks, TPMs, and cryptographic operation accelerators. Thus OP-TEE lacks a general secure I/O pathway, forcing on-sensor data encryption as the only currently-supported model of secure I/O.

Pearl-TEE [33] is a TrustZone OS developed that is focused on allowing multiple, mutually-untrusting-but-trusted applications in a TrustZone context. To accommodate this they partially bring the notion of a general-purpose OS to TrustZone by extending OP-TEE with a virtual memory system and privilege separation. They also provide access to network functionality with a delegation system that forwards socket functionality to Linux. However, we note that existing OS primitives, such as process isolation and hardware access moderation, already can provide isolation between untrusting processes and have been extensively researched. Further they do not fundamentally change the target application class from OP-TEE or alleviate the resource constraints, limiting them in the same way that OP-TEE is.

Android Trusty [4] is a TEE OS derived from the Little Kernel [56]. Trusty fits into the same niche as OP-TEE of providing trusted services to an untrusted application, but in the context of Android. An inspection of Trusty's repositories reveals a similar driver situation to OP-TEE, with included drivers ending at core platform security hardware on supported platforms.

ANDIX OS [24] is an OS for TrustZone developed specifically for industrial control systems. ANDIX also adheres to the GlobalPlatform Client TEE, which imposes similar application design limitations as OP-TEE or Trusty. ANDIX claims to also support the GNU Newlib C standard library for applications. ANDIX additionally claims to be able to request services from the untrusted world, but only for the purposes of satisfying *other* untrusted world requests. In other words, ANDIX falls inside the existing architecture of untrusted applications leveraging trusted services. ANDIX makes no claims about driver compatibility or ease of porting existing drivers.

Existing TrustZone TEEs are largely implemented with service OSes which struggle to

host general-purpose applications. Work [31, 83] has remarked upon OP-TEE's constrained memory capabilities, and discussions in the OP-TEE community have acknowledged these restrictions, but not yet fixed them.

3.1.3.3 Formal Verification

Although works like SeL4 [40] have been formally verified the burden placed on programmers is likely too great. While in theory promising, there have been few successful production-type solutions delivering formal verification as a capability.

MicroTEE [38] is based on the seL4 microkernel [40], a formally-verified secure microkernel. Their claim is that by using a verified kernel, one of the primary concerns with allowing richer applications and OSes into trusted spaces can be avoided. While this approach reduces the overall kernel TCB and the amount of TCB involved in any particular component, this comes at the expense of performance and capability. To support Linux applications the kernel must either provide a compatibility layer or run Linux in a VM. Using a VM to run applications on an seL4 microkernel contains possible privilege escalations to that VM only, at the cost of potentially requiring all applications to run in separate VMs. Otherwise applications must be modified to build against the seL4 library rather than against typical POSIX-like libraries supported by FWKs.

3.1.3.4 Existing Mainstream Kernels

A main concern with running a full-weight kernel like Linux or a BSD derivative is complexity, which balloons the trusted computing base (TCB). Linux, for example, is very featureful at the cost of high complexity, which introduces a large surface area of code which can contain bugs jeopardizing system stability or correctness. In untrusted computing this concern can be mitigated with resiliency techniques like address space protection that prevent a system crash in the event of incorrect software. Trusted applications, on the other hand, are designed with heightened expectations for correctness and data security. Linux's high complexity makes it relatively trivial to perform privilege elevation attacks and thus using such a complex solution as the trusted OS is unsuitable.

3.1.3.5 Commercial OSes

Commercial OSes are harder to evaluate due to their closed-source nature. T6 [101] is a commercial TEE OS developed by TrustKernel. T6 is closed source and focused on commercial products, though they claim to support the GlobalPlatform API, as well as rich user mode libraries. Commercial products are difficult in research contexts intrinsically, as companies may want to lay claim to functionality introduced by researchers. Additionally, we cannot verify the claims or flexibility of T6 due to its closed-source and proprietary nature. Trustonic's Kinibi TEE [102] is another commercial TEE in use on Samsung devices [16] and automotive applications by their own copy. Trustonic claims to support richer applications and to leverage TrustZone hardware.

In summation, current TEE OSes suffer from at least one of: a lack of resource allocation and ownership, an inability to support familiar, rich applications, and a lack of device drivers that all stem from intrinsic design decisions that they have demonstrated an inability or unwillingness to part with. Our design, using lightweight kernels, provides these.

3.1.4 Lightweight Kernels

We outlined a few key properties that an OS for rich trusted applications should have: ownership of and capability to use the TEE-enabling hardware directly, a familiar programming model, an emphasis on minimal size and complexity, and designed to be a first-class citizen on the system that shares resources and co-exists with the untrusted kernel. Lightweight kernels can provide all of these properties to us without introducing significant obstacles.

Lightweight kernels (LWKs) are an approach to OS design that emphasizes performance, simplicity and flexibility over feature-completeness [85]. Lightweight kernels are OSes that provide C runtimes with a suite of common system calls. LWKs follow a design philosophy explicitly rejecting generalism, specializing their design to a specific application class and associated requirements. LWKs have most recently been used in the context of high-performance computing to create resource partitions on nodes to improve performance isolation, resiliency, and scalability. We recognize that this same model can be extended to trusted computing by modeling the trusted and untrusted spaces as two such partitions,

each with their own, independent resources. It is important to note that LWKs are *not* unikernels or embedded OSes – unikernels are targeted at single applications to provide highly-slimmed kernels, and embedded OSes typically provide very few, if any, hardware abstraction layers and privilege separation between OS and application. Regarding unikernels we note the evident gradient from unikernel through LWKs to FWKs as the number of supported applications grows from ~1 to all.

Central to the design of a LWK is co-existence with a FWK, usually referred to as the "service kernel," which is often Linux. This "service kernel" has been used in the past to enable delegation of complex or performance non-critical system calls to the FWK, as well as to allow management of system resources in LWK partitions from the FWK partition. Such close inter-partition OS integration will be important in empowering rich, trusted applications, providing access to non-security critical capabilities instead. This model of close inter-kernel interaction readily lends itself to the trusted partition architecture as well, such as in the Pisces co-kernel system [74].

Finally, an LWK provides many of the benefits of both embedded OSes which provide close hardware access as well as of Linux by providing a broadly Linux-compatible userspace that supports many applications with zero modifications. Although often derived from existing OSes like Linux, LWKs are relieved of the complexity and overhead introduced by Linux's many hardware abstraction systems. This means that the hardware of the underlying system is more directly accessible, giving multiple benefits. A major design difference is the general lack of hardware abstraction layers. FWKs seek in many cases to provide an abstraction of hardware, constructing frameworks that help the diversity of hardware devices provide uniform functionality at the userspace level. This decision results in broad hardware compatibility at the cost of blurring the specific details of any individual piece of hardware, a necessary consequence of abstraction. A TEE OS has a notion of security state and configuration and for this notion of system security to be compatible with many underlying TEEs it is necessary to create an abstraction, for example by creating a kernel notion of capabilities and by requiring all kernel actions to be supplied with a capability. In contrast, an LWK would "drill through" abstraction by recognizing the fact that system security on TrustZone is achieved by setting permissions on specific regions of memory, embracing the hardware

Table 1: LWKs are comparable to existing TEE OSes in terms of code complexity.

Kernel	*Lines of Code*	*Type*
ANDIX	~26,000	TEE OS
OP-TEE	~8,000	TEE OS
Android Trusty	? [a]	TEE OS
seL4	~8,700	TEE OS
Kitten	~19,000	LWK
mOS	~17,000	LWK
fusedOS	~27,000	LWK
McKernel	~33,000	LWK
FFMK/Fiasco	~27,000	LWK

[a] Trusty is implemented as various library OS components and is difficult to gauge the "core" Trusty LOC.

directly. Though this approach sacrifices platform-independence by losing abstraction, we gain comprehensibility and ease of development. Peripherals and other devices are readily and directly accessible as well. We note that a maximally compatible, platform-independent system must also be maximally abstract, which can be seen in Linux's driver framework that ties class of devices together with generic methods and requirements. Typical LWKs realize this in a similar lines of code count to prevailing trusted OSes, including OP-TEE[99], Android Trusty[4], ANDIX[24], and seL4[40] while still providing a full-featured OS, as shown in Table 1. For this comparison we consider the LWKs surveyed by Gerofi et al [25], using only the core kernel code by comparison of source lines of code [45], as reported by the `sloccount` tool [110].

3.2 LWKs as TEE OSes

In Section 1.1 the Privacy Backplane and its requirements were introduced. Here we consider the design of the TEE OS, that enables that application. As a distributed network, the backplane itself will need to establish and maintain trusted channels between nodes to

disseminate data throughout the network. As well, each node must be capable of arbitrary workloads relevant to the IoT context: databases to securely retain collected data, computer vision and AI models to process them, and conventional systems applications to manage the inter-node trusted channels, as well as other, unanticipated operators that clients of the backplane wish to deploy.

3.2.1 Hardware Assumptions

We described the capabilities of this emerging class of applications earlier, so we move now to describing the hardware we assume when discussing this work. To support this application class, a system must provide:

1. Separate hardware-enforced partitions for secure and non-secure software domains.
2. Sufficient instruction set architecture (ISA) support inside the TEE context to run a full OS and system software stack.
3. Direct access to I/O devices inside the TEE context and the ability to isolate I/O devices from the untrusted partition.
4. Configurable allocation of system resources to partitions, including CPU cores, system memory, peripherals, interrupt space, etc.
5. Hardware components needed to support trusted, secure boot and remote attestation capabilities, such as trusted platform modules (TPMs), co-processors, physical unclonable function (PUF) hardware, etc.

These requirements are reasonable and found in current TEE hardware architectures. In particular, ARM TrustZone is a suitable candidate and its widespread use in IoT/edge systems means that our hardware requirements are already met by many commodity hardware platforms [90].

3.2.2 System Co-existence

Current TEEs such as OP-TEE, Android Trusty, and ANDIX model the relationship between the two partitions similarly to the application-kernel or VM-hypervisor relationship,

where a context switch from partition A to partition B means that a CPU context switch will occur, jumping to trusted firmware that will save A's context, load B's previously-saved context into the CPU, and then jump to partition B's code to handle whatever request was made. This follows well the current model of trusted services as being available behind some sort of RPC mechanism. While a good fit for discrete service requests, this model creates problems as the number of service requests increases and when trusted services need to run asynchronously in the background. Such "world switches", depending on the architecture, usually require a call down to firmware to securely handle the world switch. This firmware interposition additionally means that service request payloads must be marshaled in some way by the firmware, potentially resulting in unnecessary copies and a longer control path.

As the responsibilities of the trusted partition grow, confining trusted processing to explicit service requests will not be enough. Periodic wake-ups to service ongoing work, multiprocessing, and other traditional time-sharing techniques are necessary to support general-purpose applications. However, the existing TEE model does not sufficiently support this use-case. We instead fully embrace the notion of horizontal system resource partitioning. A LWK TEE OS occupies its own resource partition on the system, fully owning those resources and managing them as it sees fit, including regions of memory, CPU cores, as well as its own share of the interrupt controller and any relevant devices. This requires that the existing FWK OS be capable of being configured without knowledge of some fraction of system resources, as well as willingly ignoring that hardware which cannot be fully hidden from the FWK's view.

In other words, a trusted OS must be able to "co-exist" with the untrusted OS, echoing previous work done on multi-kernel environments [85]. The responsibility of a kernel to provide hardware functionality to userspace requires the maintenance of valid hardware state. Consider interrupt routing: each interrupt must be configured for security state, which core to route to, and what priority it has. The naive approach means that the final configuration of the interrupt controller is order-dependent, with the later configuration overwriting the previous. This can also lead to *active* contention over configuration, each kernel attempting to "recover" from what it perceives as an invalid configuration. Further, not all transitions between arbitrary states of hardware are valid and this active contention

could lead to crashes. This demonstrates two problems: even ordinary capabilities like interrupts are subject to contention, and this contention can fully impede the intended functionality. As the single-kernel model is dominant, kernels behave "greedily," configuring available hardware as the sole authority, when the way to avoid contention is for kernels to act instead conservatively.

3.2.3 Partition Boundaries and Messaging: the Interkernel

A critical topic of discussion is how the trusted/untrusted partitioning is enforced and crossed to request and fulfill services. Conventional TEEs employ one of several strategies, including memory-range protection, firmware intercession, and internal device configuration. Our LWK TEE design uses the same underlying hardware, but promotes the trusted partition to an equal partner in the system design. Rather than being managed by the untrusted OS which is moderated by the secure, trusted firmware, we treat the partition boundary similar to existing multikernel approaches, which blends traditional single-kernel, multi-core processing with remote messaging strategies, giving a model that resembles traditional inter-process communication between two applications. Explicitly, our design supposes the two partitions are able to directly message one another, bidirectionally.

Allowing the untrusted partition to initiate a message exchange is not without danger and must be carefully considered to avoid information leakage or allowing an attacker to modify control flow. Thus, the interkernel messaging channel cannot be as general as a typical kernel's "cross-call" mechanism where arbitrary functions can be triggered by one core messaging another core, a ubiquitous operation for multicore kernels today. However, a robust and generic message channel is required to support the wide variety of message types and payloads that applications and drivers may want to exchange. Any underlying channel must be agnostic to its use and capable of transmitting arbitrary amounts of data without knowing anything about that data.

3.3 KaTZe: The Kitten LWK TEE OS

To realize our design we extend the Kitten lightweight kernel. [79] Kitten is a lightweight kernel originally developed for high-performance computing. As a lightweight kernel its design emphasizes simplicity and the absence of abstraction layers which take decisions away from applications. This simplicity and flexibility has been leveraged in other work, including in performance isolation [75], virtualization [28], and hardware co-design [29], as well as trust in ARM platforms [47].

We developed extensions to Kitten to make it suitable as a TEE OS which we call KaTZe. These extensions include drivers for necessary TrustZone components as well as kernel components to integrate with the underlying firmware, which in our case is ARM TF-A [100]. TF-A is ARM's trusted firmware implementation that is used by many existing TEE OSes for the ARM architecture. A central obstacle with the existing architecture is that the TEE OS is treated as "ephemeral". To address this, we partition the system statically and allocate most CPU cores to KaTZe, and allocate only few, possibly down to one, cores to the untrusted OS as a service core. Further, the GlobalPlatform notion of TEEs is embedded into the design of the TF-A firmware and must be modified to reflect this new architecture; fundamental capabilities are currently excluded from the trusted partition. Finally, workloads in IoT sensing environments such as AI models can require large amounts of working memory, so we allocate the majority of memory to KaTZe, confining Linux to a few hundred megabytes.

The context of our work is on middle-powered, single-board ARM computers (SBCs) such as Raspberry Pis. As opposed to higher-powered server- or desktop-class computers with advanced hardware, our target systems often have some but not all of the "typical" hardware features, or have less capable versions of them, such as replacing a fully-featured MMU with a simpler, flat region-based memory protection mechanism. On the other hand, our target systems are significantly more capable than the lowest-end of micro-controllers that typically lack hardware security features, have reduced architectural register sizes, and may even lack features like an MMU entirely. Our target systems, while lower-powered, are not expected to be power-constrained, and our aim is to enable as much data processing to be done locally

as is possible without offloading to more capable servers. In our work we have selected the Rockpro64 single-board computer (SBC), which is equipped with the RK3399 system-on-chip (SoC). We have a port of KaTZe system with Linux as untrusted partition OS on both QEMU and the Rockpro64 SBC, using the existing bootloader chain of almost-unmodified U-Boot and Trusted Firmware A (TF-A) that we modified to support a general-purpose OS. While doing our research, QEMU was not well-suited to model TrustZone hardware capabilities, so our work was done primarily on actual hardware.

3.3.1 Resource efficiency and TCB simplicity

By design, LWKs present a smaller TCB and make simpler, more efficient decisions about resource usage than FWKs, where increased functionality and more exotic requirements result in more sophisticated mechanisms. In particular, KaTZe makes several choices that make reasoning about the current state of the OS easier, as well as having a smaller resource footprint. For example, a KaTZe process's memory is pre-allocated, bound to that context, and is linear and contiguous, as opposed to Linux's highly fragmentary demand-paging system which emphasizes dense multiprocessing, memory usage, and flexibility. KaTZe utilizes a non-load-balancing, round-robin, pre-emptive scheduler that further simplifies OS state as compared to Linux's Completely Fair Scheduler. Finally, KaTZe omits the fully-developed driver system that's replete with useful and convenient abstractions to ease driver development; while these features are useful in general-purpose contexts, they are largely unnecessary for specialized, IoT contexts and introduce unneeded complexity and abstraction cost.

Linux's complexity both in kernel and driver design results in an increased attack surface and a corresponding increase in the number of common and severe attacks. [39]. We make a qualitative argument that KaTZe's simplicity improves security [2], and as evidence we present several TCB measurements that can be used to compare the complexity of different trusted kernels: lines of code in the core kernel as measured by the `sloccount` program [110], the number of syscalls the kernel exposes to userspace applications, and the overall size of the final kernel executable, shown in Table 2. The table shows that KaTZe is much more comparable to an existing TEE OS than to Linux in terms of footprint. For fairness this

Table 2: TCB measurements for trusted kernel candidates.

OS	*ELF Size*	*LOC*	*No. Syscalls*
Linux 6.6	30M	~260,000	605
OP-TEE	4.5M	~8,000	N/A[a]
seL4	574K [21]	8,700	11 [b]
KaTZe	3.4M	~18,000	97

[a]OP-TEE exposes a different set of syscalls that do not have POSIX-like functionality.
[b]Implementations of the L4 series of microkernels vary slightly in syscall counts.

count includes only the core kernel counts, excluding code that provides specific platform support and device drivers.

Importantly, ports of Linux drivers to KaTZe do not necessarily increase in complexity or size due to the absence of Linux's substantial structure and abstractions; the Kitten kernel port of Linux's I2C controller driver for the RK3399 yields ~720 LOC versus the original's ~840.

OP-TEE lacks an I2C driver to compare against, and moreover these numbers do not capture the overhead of developing applications against the OP-TEE trusted application (TA) framework, which is unnecessary in either Linux or KaTZe. OP-TEE does not model conventional userspace applications and so we skip counting its syscalls. Finally, seL4 as used in MicroTEE and the similar HYDRA [21] claims to contain 8,700 lines of code in its kernel. While these numbers are approximate and qualitative, we believe they are reasonable proxies and illustrate the point that operating system capability and richness *do not* always require the weight and complexity baggage that FWKs have. On the contrary, KaTZe provides these things in a similar footprint to existing microkernels and constrained TEE OSes.

3.3.2 Secure Interrupts

Interrupts are a critical component of modern systems, used widely for multicore kernels and to enable asynchronous I/O operations to reduce busy-waiting, and pre-emptive multitasking is the model assumed by most applications. To support interrupts in the trusted partition, the system interrupt controller must be able to enforce security policy on inter-

rupts to prevent attacks, most obviously denial-of-service attacks where an attacker sends a flood of interrupts to the cores of the trusted partition, but also "misdirection" attacks where the attacker sends spurious interrupts that trigger memory reads to force the target to read compromised data.

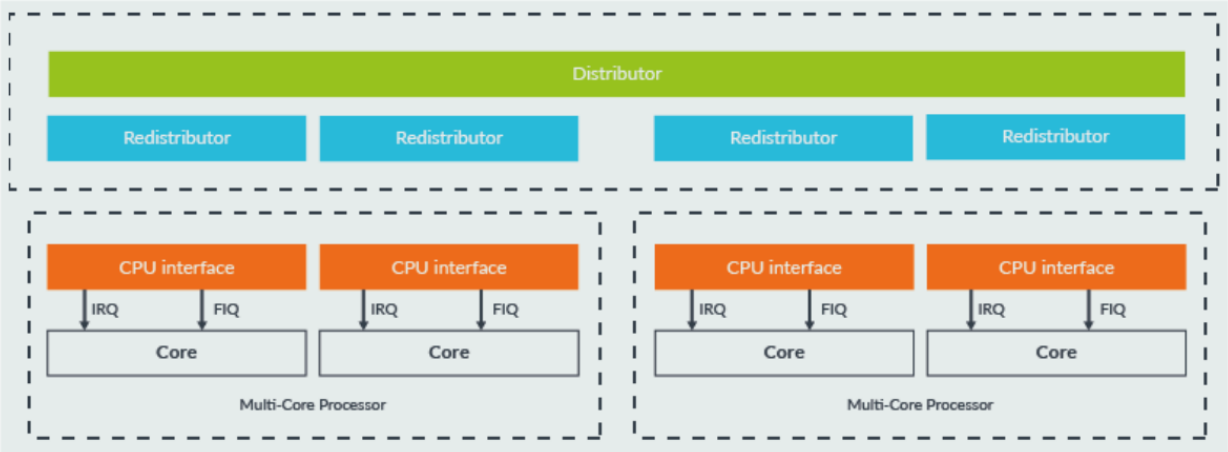

Figure 3: The GIC's hierarchical structure means the two partitions may compete for control of the GIC distributor's configuration. Diagram from ARM [7]

In the ARM Generic Interrupt Controller (GIC) architecture, devices that request service via interrupts have their interrupts routed to the system-wide distributor first, which then routes it to the appropriate core via the specific redistributor for handling. This configuration is another point of contention in the GIC, since the distributor configuration is shared state between the trust partitions. Fortunately, the GIC itself implements security state-based access controls that prevent a non-secure but privileged adversary from re-routing secure interrupts to itself. The structure of the GIC and the shared configuration state is shown in Figure 3.

The most important observation to be made is how strongly the assumption of being the only kernel is built into Linux. We describe this implementation difficulty to detail that many aspects of system hardware management are not typically shared, requiring us to rework existing assumptions. The reference interrupt controller for the ARM platform, the Generic Interrupt Controller (GIC), is TrustZone-capable from version 3 onward. This interrupt controller is widespread on the IoT class of ARM systems, including the RK3399 SoC we use. The GIC's structure is hierarchical, with a single distributor at the system level and a redistributor at each core. Per the specifications the GIC presents a different

configuration register to untrusted kernel software for configuration, but in our prototype model we unexpectedly found that the untrusted partition Linux kernel was able to read and write bits in the register that should only be accessible to the trusted partition. As a result, we were required to patch the Linux kernel to avoid resetting the GIC distributor configuration, as the trusted partition initializes before the untrusted one. During this startup KaTZe configures the GIC distributor with a configuration, that, if cleared, is invalid, which is described by ARM as an "unpredictable" state. In other words, undefined behavior. However, the existing Linux GIC code assumes that there is only one security state on the system [117]. To remedy this we apply a 2-line patch to the Linux GIC code to prevent this.

3.3.3 The Interkernel Channel

To implement our cross-partition message-passing system, the "interkernel," we opt for a straightforward design using two unidirectional memory buffers with a write lock. The size of each buffer may vary independently and the design supports concurrent messages by configuring the number of buffer pairs. As the two endpoints of the channel reside on different cores, we supplement the write lock with interrupt signaling to avoid busy-waiting on the lock itself, for which we use an inter-processor interrupt (IPI). Both buffers must be assigned to the untrusted partition, as the version of the TrustZone controller present on our development system does not support access properties on protected ranges, though this feature does exist in newer versions of the TrustZone controller [6]. This would further improve security by allowing write-but-not-read permissions for outgoing data in each direction, as well as read-but-not-write permissions for incoming data. However, this is an optimization outside the scope of this work.

At minimum our signaling system requires a single number assignment from the hardware, though we could use individual, unique numbers as an optimization. This provides compatibility with essentially all IPI-enabled platforms. On platforms like x86, this optimization is possible. However, on GIC-equipped ARM platforms IPIs cannot have an arbitrary interrupt number, and instead can use only a limited range; 8 hardware IPI numbers are available. Currently, Linux uses 7 of these interrupts for its own purposes, presumably for

the exact optimization mentioned. Unfortunately, Linux on the ARM architecture does *not* allow numbers in this range to be registered by kernel modules. The remedy for this is one of the two patches we make to the Linux kernel over the course of this work. We patch the kernel so that our interkernel channel message handler takes up the last IPI number, and we multiplex interkernel services on this number. This limitation represents an opportunity for hardware to more explicitly support tighter multi-kernel integration, since more IPI numbers would allow more services to be directly identified by their interrupt number and thereby eliminate the need for the kernel to access memory to determine which handler to invoke.

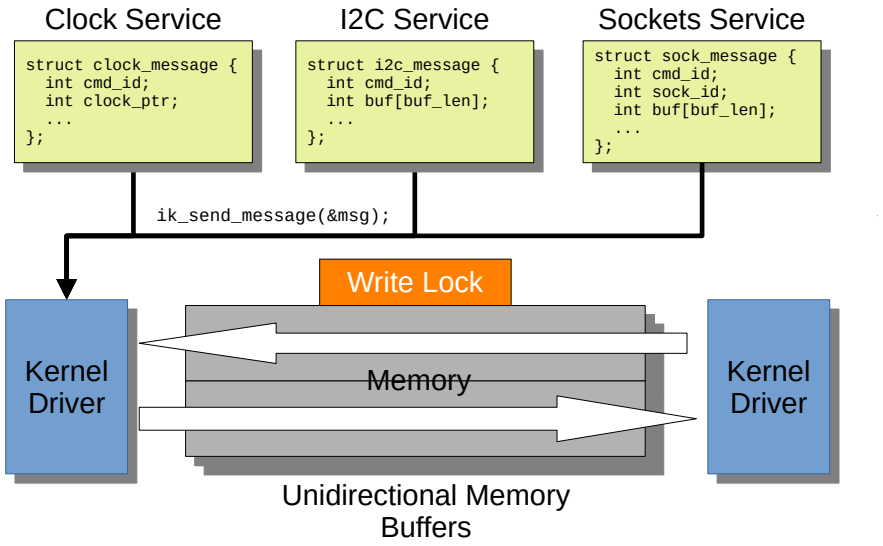

Figure 4: The interkernel provides a generic message-passing system to implement arbitrary services on top.

The design of the interkernel channel and its services is shown in Figure 4. The channel is agnostic to the messages passing across it, and we use it to implement services needed to support security non-critical functionality. We have implemented a simple memory mapping functionality that allows Linux and KaTZe processes to share memory, as well as the much more substantial socket service. As the interkernel allows arbitrary messages between the kernels, it can be used to delegate driver functionality from KaTZe to Linux when the security of that functionality is unimportant, which will be discussed in later chapters.

3.3.3.1 Sockets

Socket-based communication is supported for KaTZe applications. The socket service acts primarily by forwarding messages to the corresponding Linux kernel socket functions, as shown in Figure 5. Though this exposes the contents of all socket calls to Linux, this apparent loss of security is acceptable thanks to our "armoring the data" design focus which requires us to recognize that even memory channels are shared with potential adversaries and thus are untrustworthy. Indeed, the interkernel channel is seen analogously to how a typical OS might see the network socket: the end of the "safety perimeter." The remedy for this in our case is the same as in the wider networking case, which is to use a protocol like TLS to establish a secure channel using application-specific cryptographic identities. Socket asynchronous operations are also supported, which paves the way for listening servers on KaTZe, including web servers. As with ordinary, intra-kernel sockets, a KaTZe program and Linux program can communicate using the socket system. Though this relaxed application composition softens the barrier between trust partitions at the application level, we recall the view that the two partitions are like two networked machines; it is not part of an application's purview to know whether a remote partner is running in a compromised environment, only that the application can attest its own identity.

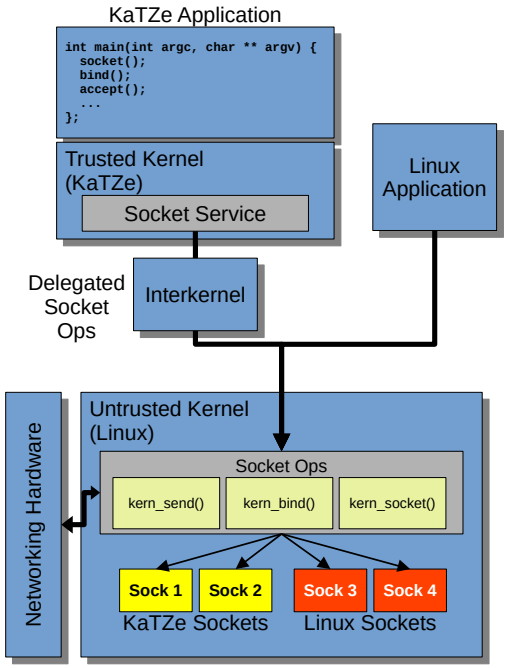

Figure 5: KaTZe provides socket operations by delegating them to the Linux kernel.

3.4 Evaluation

As previous work has demonstrated that the Kitten kernel reaches performance parity with or exceeds Linux [29, 75] on the ARM architecture [47] in many applications, we focus our evaluations on demonstrations of capability as well as the performance impact of TrustZone security controls. We investigate the difficulty of running typical IoT application stacks. Exhaustively testing all applications is of course impossible and so we aim to cover a large portion of the possible cases by targeting a "LAMP stack" of applications to demonstrate general support for applications. Comparatively specialized applications such as computer vision model frameworks are also supported, which supports KaTZe's generality.

3.4.1 TrustZone Performance

To examine the possible overhead that utilizing access control had on performance, we ran several benchmarks aimed at quantifying the memory performance overhead. We used the STREAM [61] and RandomAccess (GUPS) [44] benchmarks, commonly used for memory performance evaluations in the HPC community. STREAM tests sequential memory bandwidth, while RandomAccess tests memory updates per second. STREAM was run with an array size of 600,000 to saturate the RK3399's 1MB L2 cache. STREAM results in Figure 6 show no appreciable difference in any of the subtests. RandomAccess was run with an array length of 64M, yielding a 512M-sized array, results shown in Figure 7. These results also demonstrated no appreciable difference when TrustZone memory protection was enabled.

3.4.2 Application Support and Environment Familiarity

KaTZe provides a Linux-like environment with broad application compatibility, and KaTZe supports both statically and dynamically linked applications. While some advanced syscalls and kernel functionality are not available, many relevant applications run unmodified, and existing, popular IPC mechanisms are available. To demonstrate this, we examine application classes relevant to a trusted IoT setting, including databases, web servers, ML libraries, and WebAssembly runtimes. We describe the support that each class of appli-

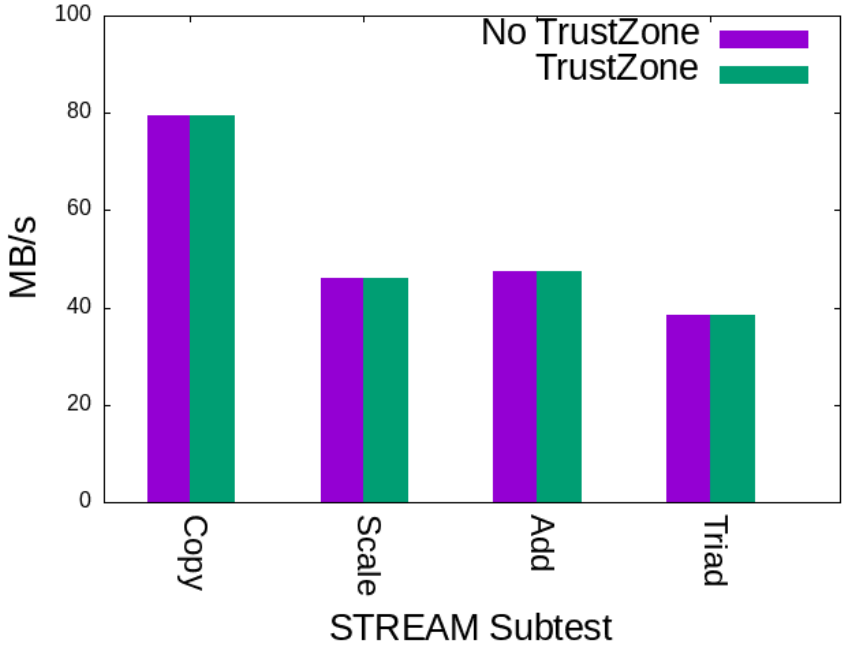

Figure 6: TrustZone memory protection imposes no overhead in the STREAM benchmarks.

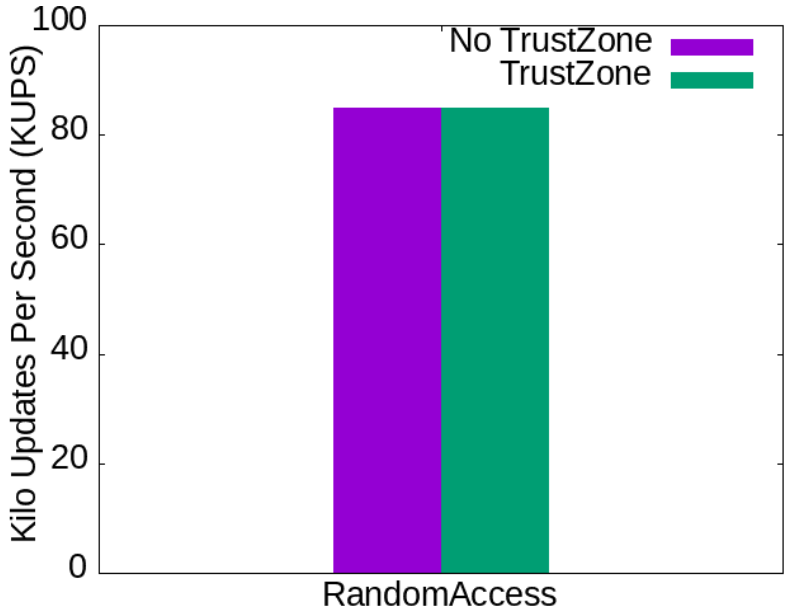

Figure 7: TrustZone memory protection imposes no overhead in the RandomAccess (GUPS) benchmarks.

cations requires from the OS environment and compare the support that KaTZe provides to that which Linux provides as well as existing ARM TEE OSes. To represent each class we selected an application suitable for the embedded/IoT environment. We examined the SQLite3 database system [96], the Mongoose web server [17], the SOD vision library [82], and the wasm3 WebAssembly runtime [104]. While not exhaustive, these applications are in general more lightweight than peers in their classes, but are still used in practice.

3.4.2.1 Mongoose Webserver

The Mongoose [17] webserver is a highly configurable, adaptable webserver suitable for use in embedded environments, but is nonetheless well-featured and capable. Designed for embedded environments that often have widely different capabilities at the OS level, Mongoose can be configured with or without support for many features, such as what underlying event polling interface to use and the underlying socket implementation. The principle demands Mongoose makes on the OS are indeed the event polling and socket implementations. Supporting this required a `poll()` implementation to extend our cross-kernel socket implementation. With this support, Mongoose works without further configuration. We describe Mongoose's performance on KaTZe below in Section 3.4.3.2. We describe our use of Mongoose in a complete application example below in Section 3.4.3.

3.4.2.2 SQLite3 Database

SQLite3 [96] is a widespread database system that can persist databases to files, being much simpler than other fully-featured database programs. SQLite3 describes itself as appropriate for embedded situations where traffic is in the neighborhood of 100,000 hits per day. Given the context of IoT sensing we expect this to be a comparable amount of requests satisfied, as this allows for a single-digit number of requests per second. For database persistence, SQLite3 requires filesystem support, similar to any other persistent database. Additionally, SQLite3 uses the filesystem to implement concurrency control, but can be configured with many different file locking methods. Again considering the low-concurrency nature of IoT, we configured SQLite3 with `flock` support, which is a simpler POSIX file-locking interface that

operates at the file granularity. With an existing filesystem layer, extending KaTZe's filesystem support for file-locking was straightforward. We describe the performance of SQLite3's included benchmark, speedtest1, below in Section 3.4.3.1. We describe our use of SQLite3 in a complete application example below in Section 3.4.3.

3.4.2.3 SOD

SOD [82] is a computer vision framework targeted at embedded contexts that also has a broad feature support. Preliminarily we used `strace` to see the syscall requirements for each and confirmed that KaTZe supports the necessary calls. We used previously-captured photographs which corresponds to a backplane node sending a captured frame to another, more powerful node. We tested SOD's CNN capability using the `tiny20.sod` model weights and the `:fast` model, with our runs having a memory footprint of 181 megabytes and yields identical object detection results as on Linux.

3.4.2.4 Mmap-based IPC

We also implemented mmap capabilities that allow an untrusted application and trusted application to directly share a region of memory, in this case the interkernel buffers themselves, which is realized as a Linux kernel module and Kitten kernel module. On top of this we implemented a simple ping-pong application as a test of its functionality. These results demonstrate that relevant IoT applications can be run without a drop in performance and either without modification or with minimal modification. Further, we demonstrate that flexible IPC mechanisms are possible with LWKs, providing alternatives to the existing trusted service architecture. We believe the examined applications constitute a class of relevant applications to lower-end IoT sensing and AI workloads and help make the case for LWKs as performant workload systems.

3.4.2.5 WebAssembly Runtime: wasm3

WebAssembly [32] is gaining popularity as a solution for security, attestation, and application portability [68, 93]. Many WebAssembly runtimes emphasize security features like sandboxing and capability management, with research using these features to provide security inside TEE environments, specifically work like WaTZ [68]. WaTZ details how a WebAssembly runtime can be extended to provide remote attestation capabilities of running WebAssembly blobs. To demonstrate the generality of our approach we ran the popular wasm3 runtime [104]. We built SQLite3 for WebAssembly and ran the speedtest1 benchmark again to demonstrate the runtime's functionality.

3.4.3 Prototype Application and Benchmarks

KaTZe supports many relevant applications without modification, and many others with a small compatibility layer provided by the OS. To show this we identified several classes of applications that are relevant to trusted IoT workloads and examined what kind of support they require of the underlying OS, such as networking, filesystem, or advanced syscall support.

To demonstrate this we built an integrated example leveraging several of these. We built an individual node that would be part of a distributed application described in one of the use cases above, as depicted in Figure 8. In our system, we have a multiprocessing stack, with a sampling application that samples the sensor at regular intervals. The sensor samples are inserted into a SQLite3 database instance. To provide access to the database we used the Mongoose webserver. In our prototype, curl on the co-tenant Linux kernel is used to submit SQL queries to the webserver which are sanitized before executing. The results are sent back in JSON format. As the underlying socket system is managed by Linux, we are not limited to intra-node communication and this service could be exposed to the broader network.

This prototype demonstrates that typical, sensor-enabled IoT workloads are possible without substantial re-engineering work. With support for scheduling, filesystems, sockets, and secure I/O, KaTZe enables this secure sensing modality.

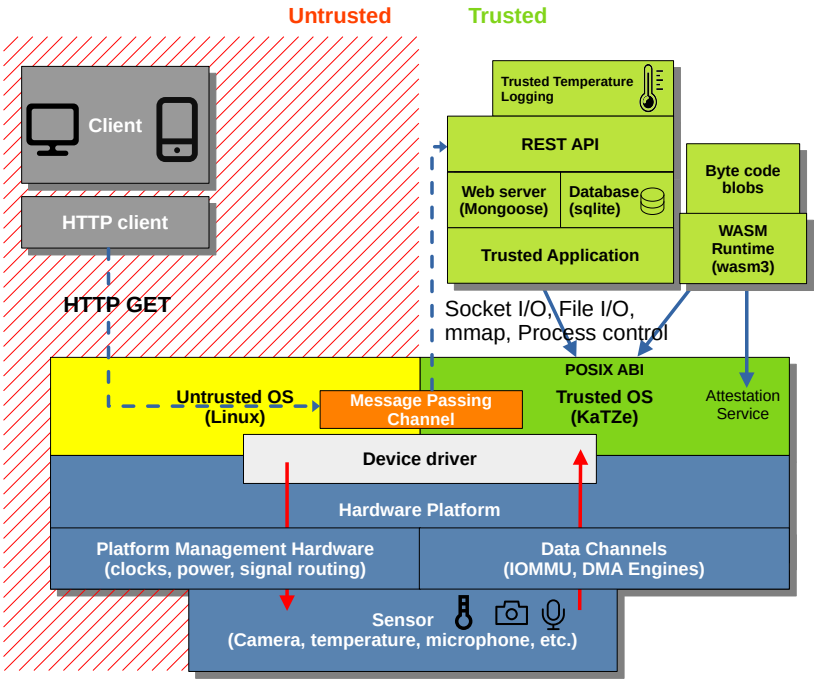

Figure 8: KaTZe supports a typical POSIX ABI "LAMP stack," using split drivers for sensors and delegated sockets for networking.

3.4.3.1 SQLite3 speedtest1

SQLite3 includes a benchmark program called speedtest1. While not as exhaustive as benchmarks like TPC-H [98], we are primarily concerned with the performance difference between KaTZe and Linux. Considering this benchmark is used by SQLite3 themselves to identify performance and feature regressions, we feel it is an adequate benchmark for comparing the two OSes. We ran the speedtest1 benchmark program on both Linux and KaTZe, both using an in-memory filesystem. These tests were run with the following parameters that specify the size the test: `--shrink-memory --reprepare --stats --heap 10000000 64 --journal wal --size 5`. We also ran the benchmark with `--size 50` to evaluate it at a larger database size. The results are shown in Figure 9. These results demonstrate that KaTZe can take advantage of the same application-level optimizations that are available to Linux, as in most subtests we observe no performance dropoff. The only test where Linux substantially outperforms is test 310, which is 5000 four-way joins.

3.4.3.2 Mongoose performance: sockets

To profile the performance of our interkernel socket implementation that provides Linux socket capabilities to KaTZe-hosted programs, we constructed a few simple benchmarks measuring request latency and raw throughput. To measure latency we time how long it takes for curl on Linux to receive a reply from Mongoose on KaTZe. To measure throughput we time how long it takes for curl on Linux to fully receive a 1MB and a 10MB file served by Mongoose. The results are shown in Figure 10. Connection latency is low at an average of 5.4ms, the average 1M transfer time is 0.69s or around 1.5MB/s, and the 10M transfer average time is 7.41s or around 1.41MB/s. Our current implementation of the interkernel channel uses a comparatively small buffer for transfers at only a single 4KB-page, meaning that transfers take many rounds.

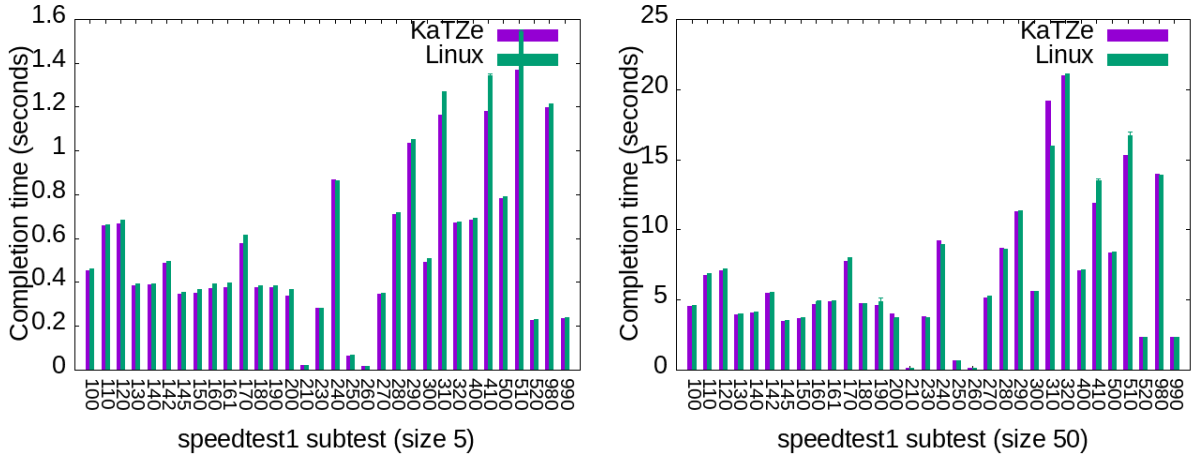

Figure 9: KaTZe benefits from the same application optimizations that Linux does for SQLite3's speedtest1 benchmark.

3.4.4 Conclusion

This evaluation demonstrates Research Insights 1 and 2, that a TEE OS can provide rich application support without serious performance penalties. By splitting the difference between feature-completeness and maximum security, LWKs can satisfy the requirements posed by rich application stacks such as the privacy backplane.

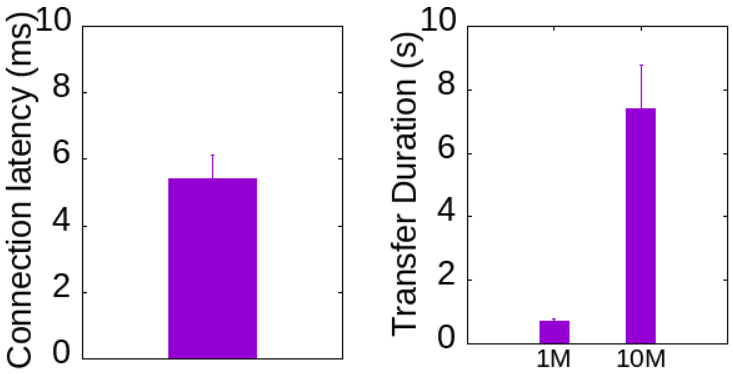

Figure 10: KaTZe's socket implementation provides reasonable networking performance both in connection latency and throughput with full delegation to Linux.

4.0 Secure I/O Stack with Lightweight Kernels

Providing secure access to I/O resources is a complex issue within trusted computing. The various combinations of the division of responsibilities between the trust domains, the available TEE hardware, and the architecture of the system software means there is no one correct approach to I/O. On some platforms providing direct access to I/O peripherals is impossible due to the architecture of the TEE hardware itself, but in other places the hardware supports it but the predominant architecture of the system as a whole, i.e. the relationship between the TEE-enabled partition and the untrusted partition, stand in the way. The split of responsibilities further muddies the waters because the hardware platform must support secure partitioning of resources in the first place. We aim to show that current TEE platforms provide adequate architectural support for a direct access model, wherein the trusted partition directly owns the needed hardware resources. Further, this is possible without simply "re-implementing" the complexity and bloat of mainstream, general-purpose kernels that already support these devices. We evaluate this chapter's work in terms of functionality instead of performance by porting several existing Linux device drivers needed for secure I/O on SoC-based platforms and then testing them using a similar image recognition pipeline, but now on KaTZe-captured images within the TEE. The approach we use in the evaluation demonstrates a general capability to borrow drivers from Linux and is applicable to many other SoC-based platforms and on-board devices.

4.1 Background and Related Work

The basic problem that we face regarding I/O is how to establish trust that the data acquired by the I/O device, be it a sensor or network device, is confidential and integral. As discussed in Section 2, in the context of multiple software partitions on the same hardware, availability is hard to achieve–without an interposed hypervisor or something similar there's nothing stopping the untrusted partition from turning off the machine as a trivial availability

attack. As a result, our main focuses are on confidentiality and integrity. To establish a secure channel there are, in principle, only two options:

1. Using a hardware channel with native security support, or
2. creating a virtual hardware channel that implements some kind of security.

The first option can be described as "armoring the wire" and in our situation refers to preventing access to the bus carrying sensitive data by partition enforcement hardware and policies. The other option is to encrypt data in some way, with the best-known and most obvious application of this being TLS/SSL connections–given the prerequisite of valid identities, a secure channel can be established on top of an insecure one.

4.1.1 Protecting the data

Seemingly the more preferable option, protecting the data itself is difficult to achieve on a local node. This approach requires each conversation party to be able to send arbitrary messages, sending the ciphertext of the message instead of the plaintext. That is, we must use *specialized* devices that are capable of encrypting outgoing data as it travels over the untrusted bus, such as McCune's work on the Bumpy system[64]. This creates a second problem of key provisioning: how to establish identities. In the case of an untrusted, adversarial hardware co-tenant, this would require a "secure provisioning" step that occurs before deployment by a trusted third party, remembering that the worst case is when the operator of the system is adversarial and seeks to subvert system security with the use of untrusted system partition. Thus, this approach is not generally applicable to a SoC environment where we seek to protect the data on system busses from other, on-chip adversaries. Some systems have taken a hybrid approach, such as HETEE [116], where encrypted messages are sent to a secure hardware enclave that temporarily takes full control of a GPU to provide secure acceleration. This differs slightly from what we have described here in that while the *message* is encrypted, once in the enclave the message is decrypted and passes in cleartext over the PCIe bus between the enclave and GPU itself. Fidelius [22] models a system with these "smart peripherals" with its man-in-the-middle technique whereby single-board computers are interposed between I/O peripherals like the keyboard and display and act as

the "security controller" for that device on its way to the main computer's SGX hardware enclave. Finally, Graviton [105] augments the GPU architecture itself to enable encrypted communication.

4.1.2 Protecting the channel

The more common approach to establishing trusted data flow within a single hardware system is by protecting the channel, i.e. policy-based, hardware-enforced access controls. These can take various forms depending on the resource they are protecting, including page table attributes, hardware that exposes different functionality depending on security mode/status, and direct intervention on interconnect transactions. This heterogeneity leads to a rather large variety of implementations depending on the particular problem and hardware platform.

Weiser's SGXIO system [108] provides a secure I/O path to applications running on an untrusted OS by exposing *virtual devices* which issue requests to SGX I/O enclaves which control the devices. This approach reduces the TCB by leaving the user application out of the trusted partition, but introduces a virtualization overhead at the same time. They acknowledge that a fully-trusted OS is yet-unachieved, largely because of the architectural limitations of SGX itself, which are not present with TrustZone.

ARM TrustZone presents a more flexible security envelope as compared to Intel SGX. The relaxed architecture of TrustZone means that a complete OS can run within TrustZone with comparatively little modification and that secure OS can directly own devices and run user applications, in contrast to SGXIO's untrusted OS with trusted driver enclaves.

As existing work shows, a device's dataflow must be wholly owned by a trusted actor to be considered trusted. The main questions are whether the device is self-owned, as securing the data suggests, where the device itself is the trusted actor, and whether the pathway for trusted data passes in some way through the untrusted OS. Current work largely takes the approaches of SGXIO or OP-TEE, where applications live in an untrusted space and issue requests for service to trusted enclaves, which may own devices.

Principally, what we are suggesting is that the trusted actor in this case be a full OS which

wholly owns the devices *and* directly hosts user applications. This scheme is necessary to provide Guarantee 1 from the threat model, that trusted software is always an intermediary between a sensor device and the untrusted world. Existing service-type OSes typically own as little hardware as possible to reduce attack surface and thus preclude themselves from hosting unmodified, general user applications. Our proposed architecture is shown in Figure 11.

Moreover, TrustZone's architectural capabilities often invites dismissal of the I/O problem; the hardware is capable of controlling access to devices, and fundamentally nothing prevents a TrustZone OS from using these capabilities to provide I/O, so many TEE OSes consider I/O to be a secondary concern. Mainstream projects like OP-TEE or Trusty can, in principle, support devices, but works [31] have noted that despite considerable project maturity, few public commodity device drivers exist. Further, implementing drivers for the wide range of devices exposed through current SoCs would greatly expand the size of the TCB and the quick development of SoCs would see any trusted OS that attempts to support all relevant platforms approach Linux in complexity. Additionally, existing and successful TEE architectures for other platforms than TrustZone, such as Graphene-SGX [18] do not address the I/O problem, as Intel SGX did not provide the ability to directly protect I/O assets.

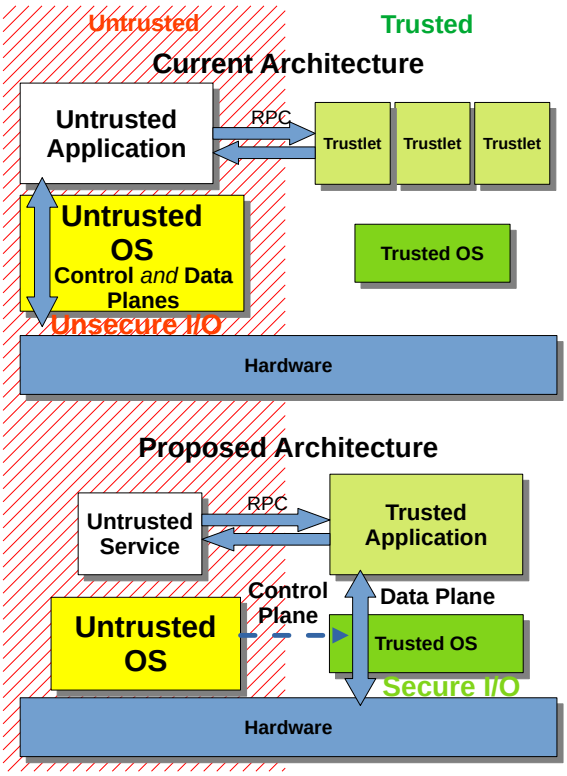

Figure 11: In our proposed architecture sensor data is securely sent to the trusted application.

4.2 Trusting I/O: Secure Driver Stacks

Fully realizing the potential of trusted, general-purpose computing requires direct, secure access to peripherals. Applications like computer vision require continuous video datastreams, where even relatively low resolutions such as 800 by 600 will reach sustained rates in the MB/s, a non-trivial load for the SBC-class device. To secure the datastream, the underlying system architecture needs to have the following properties:

1. The hardware architecture must support discriminating between trusted and untrusted transactions on the relevant bus.
2. The hardware architecture must support policy-based enforcement on relevant busses.
3. The performance of the bus must not be significantly reduced by utilizing the hardware architecture's trust capabilities.

The last requirement precludes the use of certain TEE architectures, such as Intel's SGXv1, thanks to the in-place memory encryption that protects the trusted partition's memory from bus-level attacks, but substantially limits the available memory footprint and performance [12]. Architectures like TrustZone [81] or the proposed RISC-V Keystone [50] and Penglai [23] systems that are capable of transparently discriminating, i.e. with no overhead, support the KaTZe system.

Even determining whether some hardware is compatible is not always straightforward: vendors generally are reluctant to provide public device drivers for their TEE hardware, arguing that releasing this source would be a security concern or alternatively that they constitute a competitive advantage they don't want to relinquish. [84, 65]. Whatever the reason, the extant TEEs have little in the way of publicly-available device support. As rapidly-evolving commercial products, OEMs are not particularly motivated to divulge every detail about their hardware to curious hackers and academics [66]. Generally documentation is of poor quality and can be limited to vague, brief descriptions of registers [86]. Sometimes documentation about hardware is completely absent in the case of smaller, more specialized devices like on-chip hardware blocks. Yet still sometimes documentation exists but is not distributed for public consumption, only accessible to those with an NDA, leaving only

the OEM's provided Linux driver as the sole reference for the device [19]. These poorly-documented Linux drivers are a substantial hurdle for IoT systems to create secure I/O paths to devices. To establish a secure path, trusted OS developers must either write device drivers from scant or no documentation, or port the existing drivers. For simple devices like UARTs this is not a huge hurdle, but typical IoT platforms today often include complex on-chip hardware like multi-stage image pipelines, cameras, and GPUs.

Perhaps the main advantage that a FWK like Linux has in this regard is its abundant inventory of device drivers. Many device manufacturers *do* supply drivers specifically for use with Linux to improve with adoption and continue to provide support for these devices by assigning them to open-source software support companies like Collabora and Linaro. In cases where the manufacturer does not provide the source, they may still supply a driver, as is the case with NVIDIA's GPUs for which a binary blob driver is supplied. In other, more fortunate, cases the manufacturer initially supplies a closed-source driver but later recognizes the utility in supporting an open-source one, as was the case with ARM's Mali GPU drivers.

Fortunately, it is possible to leverage previous work of the community by re-using existing Linux drivers. There are two cases to consider: closed- and open-source drivers. In the case of closed-source drivers, the target kernel must emulate Linux at a fairly low level and with fairly high fidelity. Further, re-using built drivers means emulating version-specific behavior, which can include arbitrary dependencies on other kernel systems. As a monolithic kernel, Linux drivers are *integral to* the kernel, and although Linux overall has trended toward design decisions that force kernel drivers to use specific, driver-specific kernel interfaces, drivers still have enormous power and access. The deep layers of hardware abstraction present in the Linux device framework suggest that any kernel that could support an arbitrary Linux device driver will resemble a hypervisor hosting a Linux VM to drive these devices, an approach that has in fact been used to forward-port drivers for critical legacy devices [51], which is detailed in Figure 12.

Now we turn to open-source device drivers. However, open-source does not mean good or available documentation. While a driver may be provided, there may be little to no supporting documentation to ease the porting process, such as technical reference manuals or code comments. In many cases, vendor-provided drivers have "magic numbers," referring

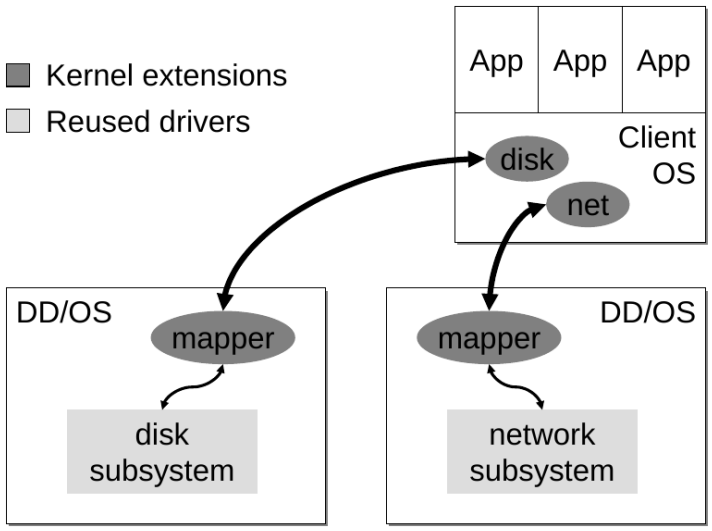

Figure 12: Re-using drivers can be achieved with the "sledgehammer" approach that hosts a complete VM, here denoted "DD/OS". Figure from [51].

to opaque values written to device registers which may themselves be undocumented. This can be done for benign reasons like loading an initial state after reset, encoded in the driver as an opaque bytestring. However, this practice can also be used to obscure device functionality, such as hiding the location of sensitive debug registers or limiting descriptions of functionality to NDA-exclusive manuals, which often pose problems for non-industrial entities to comply with.

4.2.1 Devices and internal complexity

Devices on SoC platforms are characterized by substantial internal complexity. Although some devices are simple, with control and data on a single, low-bandwidth bus that can be polled at a comparatively low frequency to function in a trusted-sensing environment, most are not. For these simpler devices the "lift-and-shift" approach where as little code as possible is changed is appropriate here, given the fewer hardware abstraction layers between the raw device and the Linux userspace. By contrast, sophisticated devices such as cameras and image signal processors (ISPs) can be very complex, with complex internal topologies

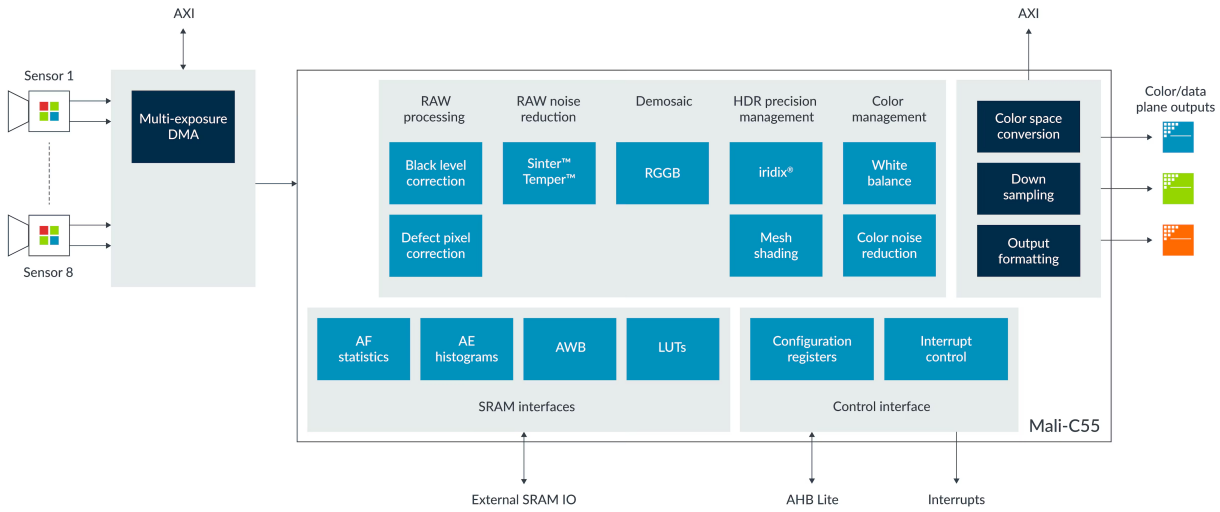

Figure 13: Block model of a modern ISP, the ARM Mali-C55, showing considerable internal complexity. Image from ARM [8]

containing internal control and data configuration. Such a device is depicted in Figure 13. These ISPs process incoming image data from a connected camera, do some processing on it, and output it in some way, typically direct DMA into a framebuffer somewhere in memory.

This complexity is also heterogeneous; some ISPs possess image correction capabilities or dual-path image processing pipelines for lower-latency image previews to be used in "selfie cameras [86]." Likewise, many camera modules contain on-board ISP capabilities, including many off-the-shelf models for Arduino-like devices. Finally, cameras can also capture images in multiple formats, each with their own properties, requiring configurations between the camera and ISPs to agree to function correctly.

4.3 Simplifying the Driver Stack

Mainstream FWKs employ extensive device driver infrastructure to support the wide variety of devices available. Generalist kernels such as Linux have a primary goal of supporting as many kinds of devices as possible, which results in large driver frameworks. These frameworks model a given class of devices, whether it is I2C controllers, graphics processors, or even UART devices. Hardware changes over time and dropping support for a device

architecture is unpopular and thus not common [49]. As a result, we can infer that device frameworks continuously accumulate complexity in order to support new features and architectures, but only infrequently trim it. While this results in good long-term support in mainstream kernels, it means that largely irrelevant, legacy devices substantially increase the complexity of the device class a driver framework model must consider. Further, kernel subsystems over time must themselves evolve to interoperate with other subsystems–the integration between the Linux media controller framework and the `v4l2` subsystems is a good example as Madieu describes it [59]. The sum total is that this creates a maximized kernel footprint in exchange for a minimized device-specific footprint, the natural inverse of Passos's 2021 discussion of "feature scattering." [78]. The growth of complexity is also visible in the corpus of "debloating" research which aims to undo this cumulative complexity [42].

While this approach is well-suited to such FWKs, it is *not* well-suited to TEEs, explicitly due to the large amount of complexity in fully-privileged kernel code. Even if a kernel's subsystems and frameworks can be leveraged to quickly support a single device's needs, the kernel contains the complexity needed to support all the devices that *aren't* present in a given configuration. Since we assume that untrustworthy software will not be deployed to trusted partitions, what we require of the OS changes. Indeed, more functionality and hardware access can be forced onto applications, moving some driver logic that currently resides in the kernel into userspace libraries. This sort of reduction in kernel hardware abstraction layers (HALs) means that substantial amounts of code from the kernel and drivers can be removed. With this removal of HALs and shifting responsibility onto userspace libraries we can radically simplify the driver framework.

Although we seek to reduce the gap between a userspace library and the hardware we do not remove it entirely. Access to hardware should still be moderated by the kernel, though our model is intentionally flexible. We believe that a message-based model of how userspace libraries interact with kernel device drivers is ideal. Many such frameworks exist, including the `ioctl` and virtio frameworks [88]. By defining commands atop a generic message bus, hardware can both generically and safely exposed to userspace. In the `ioctl` model the kernel serves only as a secure intermediary between the driver code and userspace, which use variable-length memory buffers copied across the user space/kernel boundary as the message

bus. This mechanism allows device drivers to much more directly expose their functionality to applications, as opposed to framing a device's functionality inside the traditional UNIX file-based operations. Indeed, this concept of kernel bypass/minimization has even become popular in Linux, in turn eliciting other research kernel designs, such as the demikernel [114], which contrasts the existing performance-oriented bypass paradigm with a desire to offer higher-level abstractions.

4.3.1 Driver Complexity: Video4linux2 and the Media Controller Framework

The complexity of SoC devices like ISPs has resulted in Linux adopting a correspondingly complex framework. The video4linux2 framework, `v4l2`, attempts to support all real-time video capture devices, which includes cameras. This framework models devices as containing subdevices of their own, each potentially exposing their own operations to manage aspects of their operation, the so-called "many knobs" of `v4l2`. Additionally, the `v4l2` framework attempts to manage the entire system, including DMA and memory management [59]. Further, the `v4l2` system has integration with the Linux media controller framework, which creates a graph-oriented API of "pads" where data flows from sources to sinks. As opposed to the media controller framework, `v4l2` itself has an interface layer to userspace, the videobuf2, or vb2, interface, a comparatively simple model in which a "buffer" containing video data that may be broken up into separate planes as required by the image format. This vb2 interface connects to a queue system which manages the memory and datastreaming of the underlying `v4l2` device. The complexity embodied by these two separate frameworks, `v4l2` and the media controller framework, combined with the overlap in responsibility between the videobuf2 portion of `v4l2` and the media controller framework, means that a significant fraction of a Linux camera driver is just integration with this framework.

4.4 KaTZe Implementation of Trusted I/O Devices

We realized the above design points in KaTZe, porting and simplifying several device drivers from Linux into the Kitten kernel. We detail the implementation here, connecting decisions made to their corresponding design points.

4.4.1 Secure Device Access

Devices in ARM-based SoC platforms are nearly always and exclusively accessed using a memory-mapped I/O interface in the CPU's regular address space. Some devices support a register-based interface to support programs written for older ARM versions, but this interface is deprecated and is explicitly disabled under certain circumstances [5]. As ARM TrustZone includes a "TrustZone Controller (TZC)" that enforces security policy on transactions on the memory bus, this security hardware does double duty and acts as an I/O gatekeeper as well. In addition to securing the portion of system memory in the logical trusted partition, securing the MMIO address ranges associated with a given I/O device or controller allows us to assign a device to a specific trust partition. Using the memory interface as the "thin waist" is a common and flexible approach in TEE hardware, used in TrustZone, RISC-V designs, as well as in OpenPower.

More sophisticated versions of these controllers allow finer-grained policies over memory, including region alignment, region read-write-execute properties, and the direction in which a policy applies, i.e. untrusted access to a trusted region, or trusted access to an untrusted region. We used a simpler version of this controller in this work which requires 2MB region alignment and only prevents untrusted access to a region. A more complex version of this hardware would enable a stronger partition boundary that can protect against programming errors in the trusted partition or accelerate cross-partition message passing.

4.4.2 Underlying System Bus Architecture

The architectural interconnect is of considerable importance with regards to secure device access. In the case of our RK3399, the bus between the camera and its associated ISP

block is dedicated, meaning that control over the output of the ISP means only bus-level snooping can extract raw camera data. Knowledge of the system architecture at this level is critical for correctly drawing the partition boundaries and understanding what devices must be controlled strictly by the trusted kernel. As we will discuss later, the fact that the dataflow is immutable but control is not will be used to further reduce the driver surface area contained in the trusted partition. We note also that our design is not dependent on this property, as the bus between the ISP block and memory is *not* dedicated, it is the main system interconnect. We recognize that bus situations fall into three categories, two of which we can tolerate:

- *Fully-isolated buses, as the RK3399's camera-to-ISP bus.* The use of this device implies ownership of these devices and thus the isolation of these busses is beneficial.
- *Shared busses with access control capability, such as the RK3399 main interconnect.* This is the general case for TrustZone, where the TZC is capable of segregating secure traffic from non-secure traffic.
- *Shared busses without access control capability, such as I2C.* This is the only intolerable case, which consequently requires exclusive access to the bus. Most I2C bus controllers lack a notion of security given the typical deployment requirements for I2C-based devices. This means that an I2C bus *cannot* span the partition boundary and must reside fully within one partition.

These categories outline the general need for access control-capable busses to support the KaTZe system, a requirement met by many existing SoC-based systems.

4.4.3 HTU21D Sensor: I2C Control and Data

As a simple example case we implemented support for the HTU21D sensor in KaTZe. The HTU21D sensor is a simple temperature/pressure/humidity sensor that uses the I2C bus for both control and data. The device is well-documented thanks to its popularity in hobbyist communities and the register map and data format is readily accessible. The RK3399's I2C controller is simple, and our port of the Linux driver is ~720 LOC, versus the original's ~840 LOC, a reduction of 15%. While lines of code are only an approximate

measure of complexity, it demonstrates that at least some portion of the complexity of device drivers in Linux is owed to tying those devices into the elaborate Linux device framework. We will show in later chapters that this TCB reduction and driver simplification can be achieved with many kinds of drivers. The actual sensor driver that both fetches and decodes samples is realized in ˜150 lines.

4.4.4 IMX214 Camera: I2C Control, D-PHY Data

Our development system has the Sony IMX214 camera module. Typical for this class of modules, the control interface is exposed over I2C. The TZC allows us to gate access to the I2C controller with memory region-based access control. As mentioned above, the data from the camera flows over a bus with fixed inputs and outputs. The camera driver itself allows for multiple resolution and exposure choices, and contains configuration as byte blobs that must be written to the camera. Otherwise the driver is self-contained and interacts with the camera device solely via I2C. Thus the Linux camera driver is a "shell" through which the kernel issues targeted I2C messages. As part of the port we moved substantial portions of camera configuration logic into userspace, preferring to expose hardware instead of providing an "IMX214 camera service" as the Linux driver aims.

4.4.5 SoC Device Complexity and Documentation: RK3399 ISP

Current SoCs are highly-integrated devices, with a variety of components at the interconnect level that share common chip resources. This level of integration means that across SoCs implementation details vary, even when SoC designers re-use blocks across designs. As a result of these non-standard designs, manufacturers have great influence over what programmers and system designers know about their hardware, which is wielded for the purposes of commercial advantage or perhaps "security," as discussed shortly. Combining weak documentation with high internal complexity results in difficulties for systems like KaTZe where system decomposition at the hardware level is at the core of the design. Among the most complex devices present on a modern SoC is the image signal processor, the ISP. The RK3399 is no different, with its "RKISP1" ISP block being composed of several sub-devices,

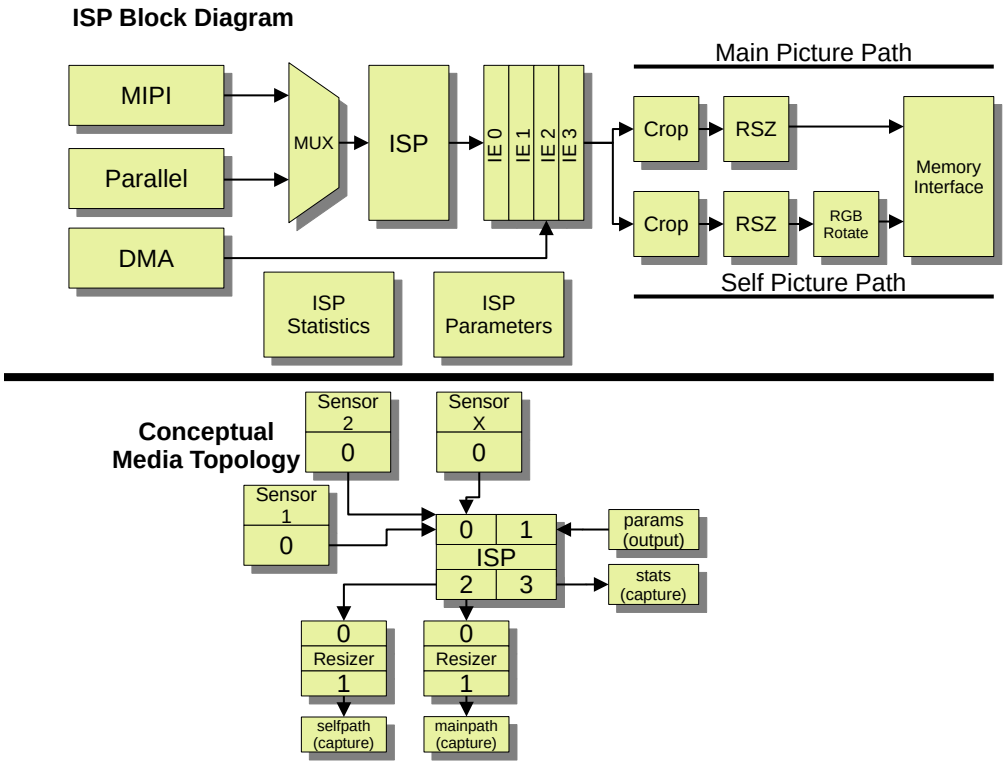

Figure 14: The logical structure of the on-board ISP is several sub-devices.

each with its own register space and configuration. The ISP has resizers, "image enhancement processors", and croppers, across two different pipelines that can output in different image formats in several different ways. Unfortunately, Rockchip, the manufacturer of the RK3399, does not make the manual section concerning the ISP itself public. Instead, they have upstreamed into Linux a driver with considerable in-line comments and associated long-form code comments, which explain the use-cases for the camera that the driver supports, which is not complete. Figure 14 shows a block diagram of the ISP, transferred from the ASCII art diagram found in the code comments.

To wrangle this complexity there are two broad options: a customized driver stack for each unique piece of hardware, or a maximized driver framework with hardware abstraction layers (HALs) that ease code re-use. Each approach has advantages: a custom driver stack simply is "hardware-oriented" and clearly communicates the hardware's capabilities, whereas a complex HAL framework is "kernel-oriented" and can shorten developer time. For decomposing system hardware, we prefer when possible to use hardware-oriented code, as systems like KaTZe are not competing with mainstream kernels like Linux to be "general-purpose"

systems that run all applications across all kinds of hardware.

The Linux ISP driver is fully "kernel-oriented" and is integrated into the `v4l2` system, which organizes each of those components into their own sub-devices. While convenient for large-scale codebases, this complexity and cleverness restructures control and dataflow into a graph which only the `v4l2` and media controller frameworks can easily understand. The Kitten kernel port of the RKISP1 driver is 3928 lines of code versus Linux's 4673, again a reduction of about 15% in the device driver itself.

From the register map in the driver we infer that the device supports some form of DMA to output finished frames into main memory, but the Linux driver doesn't support this and thus we could not support this functionality–another instance of the trend of device manufacturers obscuring capabilities. The ISP instead outputs frames in a scatter-gather style by supplying addresses of memory buffers to the device which are filled with data, at which point the device interrupts the core to request the memory buffers be changed out.

To restrict access to certain parts of memory, the ISP contains its own IOMMUs. In addition to supplying memory descriptors, the contained IOMMU must be configured correctly to describe what parts of system memory the ISP can access. In this case the IOMMU is fully embedded within the MMIO range of the ISP itself, making securing the IOMMU simpler. Other architectures, such as x86, share a single, system-wide IOMMU [36, 3]. The ARM model treats the IOMMU and ISP as independent components at the architecture level, which allows them to behave identically whether assigned to the trusted or untrusted partition.

4.4.6 Limitation: Undocumented Security Features

Security through obscurity is a technique that is unfortunately used with some regularity [66, 80, 13]. Hardware counters have been used to detect side-channels [52, 106], including undocumented ones [30]. The use of undocumented features on hardware has also made possible new kinds of attacks, such as by exploiting cache behavior [60]. Our port of the KaTZe system to the RK3399 in particular presents an instance of this difficult general trend. For the RK3399 most security features are configured by writing to a "secure general register file

(GRF)," a large block of registers exposed as a single device through which security features are configured. While part of this register file is documented through bits and pieces of published drivers, it is not fully described in public manuals.

Each device that can issue transactions onto the memory bus, or "bus masters" as ARM reference material describes them, has a security state which is managed by the TrustZone controller. In theory, this allows the system to be partitioned into secure and non-secure partitions, including most of the components of the SoC. However, this functionality is incompletely described in the manual making it impossible for us to assign the RKISP1 to the secure partition in practice. Research suggests that the manual sections are available under NDA, which we were not able to obtain.

4.4.7 Userland Camera Interface

To deliver frames from the kernel into userspace we expose functionality through the standard `ioctl` system, offering an operation to map frame buffers directly from application memory to receive the camera output, as well as ordinary copy from kernel buffers into user memory. The interface is shown in Figure 15, which we drive from userland using a simple library. In some ways this design resembles a simplified version of `v4l2`'s own `vb2` system. However, our system contains the same level of kernel complexity; the complexity must reside somewhere and we have opted to push understanding the capabilities of various camera devices into userspace, which is more flexible and easily changed as compared to kernel code.

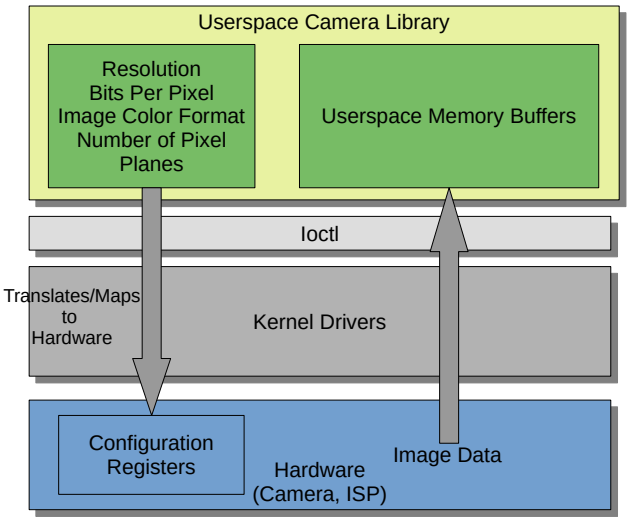

Figure 15: Our driver model configures the ISP with a userspace library which is effected by kernel drivers to hardware, without device abstraction.

4.5 Evaluation

KaTZe's assumptions about hardware combined with its kernel design allow a "lift and shift" approach to be used with many SoC platforms. We ported Linux drivers to demonstrate the viability of the approach as well as those necessary to enable a trusted I/O path relevant to a privacy backplane use-case, which in this case is a pipeline from a camera into a vision model. We first ported the HTU21D temperature sensor to demonstrate feasibility and then ported the RKISP1, IMX214, and supporting platform device drivers to enable the image pipeline. The HTU21D sensor is I2C-based, such devices being commonplace on edge/IoT platforms, and has the same functionality as in Linux. We leave a full examination of the HTU21D sensor and the broader I2C bus's performance to later chapters. The ported IMX214 driver has feature parity with Linux's, as does the RKISP1 driver, however it has additional debug and statistics features that we ported but do not use. We retained this code in our ported version to facilitate the most fair comparison possible regarding complexity. While platform devices such as the RKISP1 will vary widely from one SoC to another our porting approach is applicable to other SoCs as well. To evaluate our drivers quantitatively we measured sampling rates, but our focus is on feasibility before performance.

4.5.1 ISP TrustZone Overhead

It is documented that TrustZone has little to no architectural overhead on memory transactions [81], suggesting that assignment of the ISP to the secure transaction domain should have little to no overhead. However, because the RK3399 is under-documented and we were not able to fully realize the ISP outputting frames into secured memory, we cannot be sure that its performance under TrustZone will be exactly the same. According to the RK3399 technical manual [86] bus-level security is implemented by the ARM TZC-400 controller device and according to that controller's manual[6] *all* transactions on the protected busses are checked for security status. The RK3399 manual also shows that the bus connecting the CPU to memory is the same as that connecting the ISP to memory, so the differences in security status should affect any CPU-to-memory benchmarks in the same way. Thus we

refer to our previous memory benchmarks which test exactly this, Figures 6 and 7, which show that TrustZone enforcement status imposes no overhead on these memory transactions.

4.5.2 Image Recognition on Captured Frames

We used a simple, unoptimized image recognition pipeline to demonstrate the functionality and viability of our approach. The pipeline captures images from a Sony IMX214 camera at the native 1080p resolution, which the RK3399's ISP processes and resizes to an output size of 800x600 with an output format of YUV420p, which is a multiplanar format. The output format is not of huge significance, but in this case one of the planes is the grayscale version of the captured image. The facial recognition system we use is the SOD model [82]. SOD supports a high-speed facial recognition model type called "Realnets," which they claim are ideal for lower-power embedded devices; we tested both the Realnet code as well as the more capable CNN code. We discovered that the Realnet models do not recognize correctly on the ARM64 platform, including both the Linux and Kitten kernels, though it is functional on our x86 development machines. We verified that the Realnet model is *capable* of recognizing images captured by the camera by running recognition on our development machines, but the throughput of the Realnet code is dependent on the model itself. To demonstrate an end-to-end, in-situ solution we were forced to use one of SOD's object recognition CNN models, which has substantially longer inference times, but works on both our ARM64 evaluation platforms as well as our development machines. In both cases the userspace application is simple, mainly acting as glue between the image capture and SOD code. To show broader viability, these benchmarks are conducted on a single A53 core even though the RK3399 has the more capable A72 cores. The results are shown in Figure 16. Even though it is an unoptimized research prototype, the KaTZe stack achieves 27FPS, with Linux performing at 126FPS. The recognition model confidence scores are identical between Linux and KaTZe. These results demonstrate that, while not as performant, our argument that a simplified kernel can achieve acceptable performance while delegating security non-critical functionality by keeping only the data-critical parts of the pipeline in the secure space, is valid.

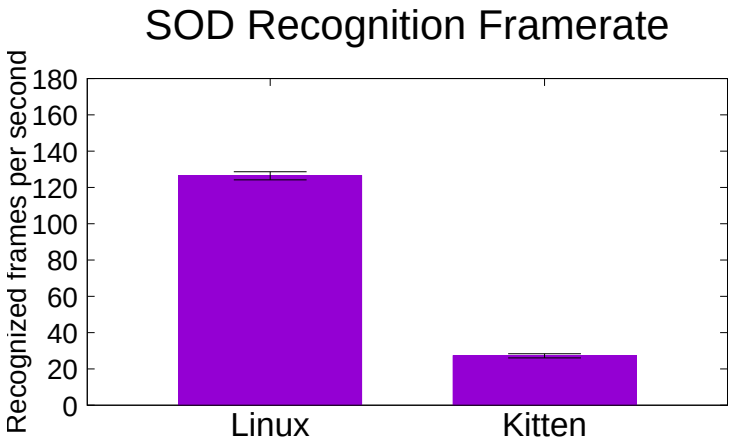

Figure 16: SOD achieves a recognition speed of 27 frames per second on KaTZe compared to Linux's 126.

4.5.3 Conclusion

This evaluation demonstrates Research Insights 3 and 4, that extending the trust boundary into all parts of the hardware when the TEE supports it provides robust I/O separation without significant overheads, another key requirement of a usecase like the backplane. Instead of comparatively complex systems that insert hypervisors to control the untrusted partition or requiring specialized hardware, thorough use of existing TEE architectures allows systems to directly secure relevant I/O paths with acceptable performance costs, which in turn allows greater processing of data near where it's collected on nodes.

5.0 Paravirtual Device Drivers

A critical design point for TEE software is the trusted computing base, or TCB. Briefly, this term refers to all the complexity that needs to be defended against attack, including both the hardware and software, and much existing research has focused on TCB size as an approximate metric of security, either for [18, 94, 50, 9] or against [12]. Unfortunately, mainstream kernels serve as evidence that as the set of drivers increases so too does the complexity of the driver infrastructure in the kernel. In Chapter 4 we described how we were able to simplify driver code when it lives in the trusted partition. However, in a shared system with both trusted and untrusted system software managing resources, there will be overlap. We show how this resource contention can be used beneficially by delegating the work of managing certain shared resources to the untrusted partition without sacrificing security. In fact, most devices on a given platform do not *need* to be fully within the secure partition and should not be. We start this claim with the observation that in a multikernel system it is difficult or impossible to create a "hard" boundary between the partitions that logically resemble two disjoint systems communicating over a message channel, given the underlying, shared hardware platform. Instead, we should make the trusted partition a client of the service that the untrusted partition already provides for itself–a functioning system. However, we must do this in a way that doesn't compromise our trusted partition's security. We intend to use paravirtualization to characterize a device as decomposable into functionalities, each with distinct security properties. Our key idea is to *extend trust to the paravirtualized I/O plane.* By adding a notion of hardware-enforced security to the paravirtual plane, we can re-use many of the ideas explored by research on paravirtual drivers by shunting non-security critical functionality out of the trusted partition and into the FWK. We evaluate this framework in terms of both functionality and performance by showing that the image pipeline shown in the previous chapter doesn't *require* the wholesale porting of all involved device drivers while also measuring and characterizing the TCB savings achieved with paravirtualization in terms of both software lines of code and hardware inside the trusted partition.

5.1 Security Sensitivity of Devices

We can discuss a device's security sensitivity by examining it across a few different dimensions:

- Does it interact with the trusted partition, either through the control of the device or its data?
- If it does interact, what impact does it have on confidentiality, integrity, and availability (CIA) properties?
- Can a device's functionality be decomposed into pieces that have differing security properties?

As a quick heuristic we can ask if the device even interacts with the trusted partition. We consider the two partitions as separate systems–if hardware partition enforcement is configured correctly, it is logically equivalent to as if they were two systems separated by an airgap. Assuming a device interacts somehow with a trusted partition, what effects could an adversary have on the trusted partition using that device? Consider the difference between a DMA engine and the power controller: an adversary that controls a DMA engine capable of interacting with the trusted partition could arbitrarily modify or exfiltrate data in the trusted partition, affecting both confidentiality and integrity. However, a power controller by the adversary can only affect availability by powering down trusted devices or possibly the whole system. Further, a compromised data bus between a specific device and the memory controller can impact availability, but not confidentiality–without the memory controller, an adversary cannot change *where* the data goes, just *if* it goes. Finally, we must consider a device's internal capabilities and evaluate each of them separately. For this we will consider an I2C controller: the controller is the gatekeeper for sending messages to devices on that I2C bus, but requires the support of other system devices to accomplish this, such as power, a clock source, and configuration of multiplexed pins. As shown in Figure 17, we can decompose the I2C controller into the separate security domains of the bus messaging capability and those supporting devices; similarly to the main system bus, the devices that support the I2C controller can also affect availability, but importantly they *cannot* affect confidentiality.

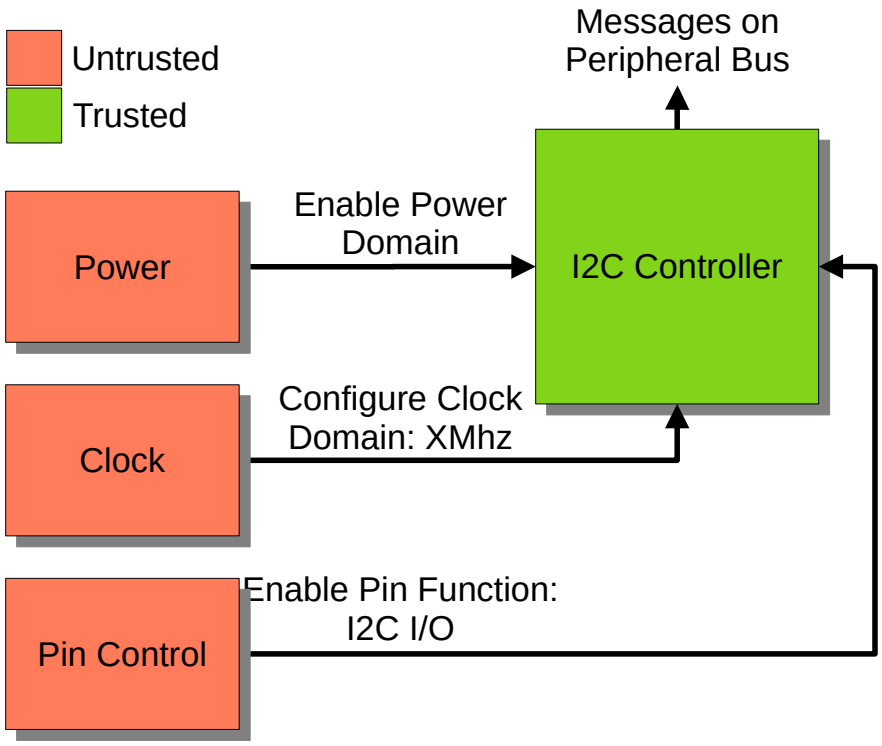

Figure 17: Decomposing a device reveals some have heterogeneous security properties which permit secure shunting of functionality out of the TEE.

However, the messaging functionality of the I2C controller impacts all three properties, such as by sending false messages (authenticity) or intercepting valid ones (confidentiality). Finally, we make an observation that certain capabilities cannot be used by the adversary without damaging themselves: the capability to power-down the CPU *could* be used against the trusted partition, but on a system with only a single CPU or without the ability to power down individual cores, this means also powering down the CPU cores controlled by the adversary. While this capability would affect availability, we can reason that the adversary cannot use it because it would also degrade their capabilities.

Now we can suggest that the functionality that cannot affect confidentiality or integrity of sensitive data ought to be left to the untrusted partition to handle, as it already knows how to do this and we are only reducing the amount of functionality we have to teach the trusted partition how to do itself. Splitting drivers across the data-control axis will allow us to leanly and securely paravirtualize many of the complex and troublesome platform devices that make up SoC-based architectures, resulting in a reduced TCB and improved trusted software

portability without compromising the logical boundary between the partitions themselves. This concept supports Guarantees 1 and 2 in the threat model, that untrusted software cannot see the data that devices handle, but may see that the devices are active.

5.2 Background and Related Work

When sharing a single hardware device between two pieces of software some concessions have to be made to avoid conflicts; different software may configure hardware incompatibly. This problem space can be described as an axis from "hardware-aware" to "hardware-unaware." For example, the filesystem abstraction fully hides the details of an underlying block device from software, instead providing an abstraction and handling the details of hardware at the kernel level, and history has demonstrated this approach to be quite practical for general, user-space software. At the other end are "hardware-aware" approaches, often in contexts where abstraction is too costly, such as embedded or real-time OS systems. To cut abstraction costs these systems sacrifice some degree of flexibility and portability by showing the software more of the details of the underlying hardware.

These approaches work well for userspace software, but what about for kernels themselves? When sharing hardware between multiple kernels the problem becomes more difficult. If kernels safely multiplex hardware between applications, who multiplexes it between kernels? Alternatively, who provides hardware services to the kernels that it can then expose to its applications? When a device owner/host kernel wants to share hardware with a guest kernel, its options range from full virtualization to co-ownership. In the case of full virtualization the guest believes it is interacting with actual hardware, but this approach introduces overheads or requires specialized hardware to avoid those overheads. At the other end is co-ownership or direct passthrough/assignment. Direct assignment is a straightforward solution in the fact that it is *not* a sharing technique, but when a device host grants full control over a device to a guest. Alternatively, co-ownership is where each kernel is aware that it shares the hardware, which introduces state coordination complexities so that each kernel does not corrupt hardware state for the other kernels. Hardware is often not designed

for this, possibly requiring coordination by a third party, such as the underlying firmware, to achieve.

Paravirtualization is a well-known virtualization technique [87, 11, 95] which strikes a balance somewhere in the middle of these two options, by requiring the the host and guest to cooperate. The guest kernel knows it accesses an abstracted piece of hardware provided by the host, and the host typically exposes some part of the underlying hardware to the guest. In some cases the guest accesses hardware through a "generic" interface that the host provides, essentially turning hardware into an abstracted service itself. In traditional hypervisor contexts paravirtualization is used to enable guest-host cooperation to achieve better performance or portability. Perhaps the most prominent device paravirtualization framework is virtio [87], which is widely enough used that cloud providers such as Amazon use it as the preferred driver in hosted VMs [15]. Paravirtualization of a device works by recognizing that data and control are separate–it is possible for a guest to request specific blocks from an underlying block device and to receive them without understanding how to interface with that specific block device's controller. This separation allows the paravirtual host to act as the control surface, allowing the guest to delegate complexity. In traditional virtualization environments this is done for a variety of reasons including scaling and performance management, as well as for portability. These frameworks expose a "generic" version of the device that the guest interacts with. In virtio's case, communication between the guest and host occurs over command ringbuffers with a scatter-gather type-indirection to point to input and output memory buffers. This structure decouples the command and its parameters from the input and output data [73].

Paravirtualization has been explored before as a technique for crossing trust boundaries [71, 115]. Zhou's Wimpy kernels model heavily uses this idea, but with the addition of a verification model mediated by a hypervisor. Their work specifically focuses on handling the complex case of USB which multiplexes control and data on the same channel, which substantially complicates sharing such a device.

5.3 Split Drivers to Reduce Trusted TCB

The split driver model, shown in Figure 18, is a key component of our paravirtualization approach. To efficiently share devices we suggest a "split driver" that logically spans partitions. From the trusted perspective, control-related functionality is mostly or wholly implemented as calls to the portion of code residing in the untrusted partition, whereas data-related functionality is implemented wholly in the trusted partition. Importantly, the device and driver model remains unchanged within the trusted kernel. With the use of a tightly-integrated cross-partition messaging channel, drivers can easily exist within both kernels simultaneously. This gives us control functionality by leveraging existing functionality inside the FWK, in-place, often with little modification. As Liu [57] and others demonstrate, paravirtualization is useful enough that even when using some degree of bypass splitting the control and data plane is warranted.

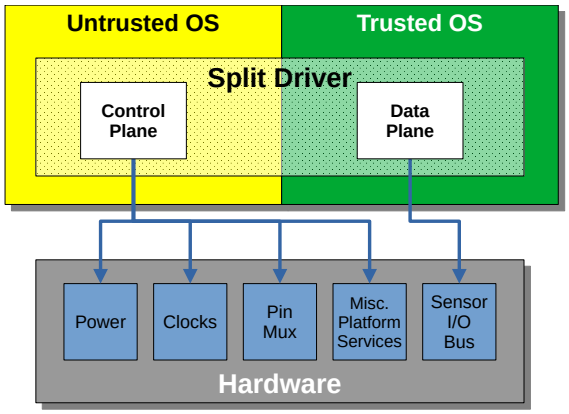

Figure 18: Our proposed driver model spans both the untrusted and trusted OS, delegating platform management to the untrusted partition.

Any paravirtual design with a notion of trust must adequately address how to securely deliver data between the host and paravirtual domains. In a hypervisor context, securing this channel so that the host domain does not leak information into the paravirtual domain is sufficient, as hypervisors can already inspect guest memory. However, in a *trusted* paravirtual driver security must be extended in the other direction as well, preventing the paravirtual domain from leaking information to the hypervisor/paravirtual host. Further highlighting this requirement is that our paravirtual context is across a horizontal rather than vertical axis and thus privileges are not hierarchical in the way that hypervisors are with guests, making existing hardware enforcement like nested page tables irrelevant. Instead, the trusted platform must provide fine-grained enforcement based on security domain, e.g. by being able to forbid the trusted partition from accessing the untrusted partition as well as preventing the

reverse. Such functionality is currently available on commodity TrustZone systems, which allows separate read-write permissions applicable both for secure-to-nonsecure accesses as well as the reverse. This hardware capability relieves the developer from the burden of proving their code correct as the only means of ensuring no data leaks between domains, which is not practical in the face of an adversarial untrusted partition.

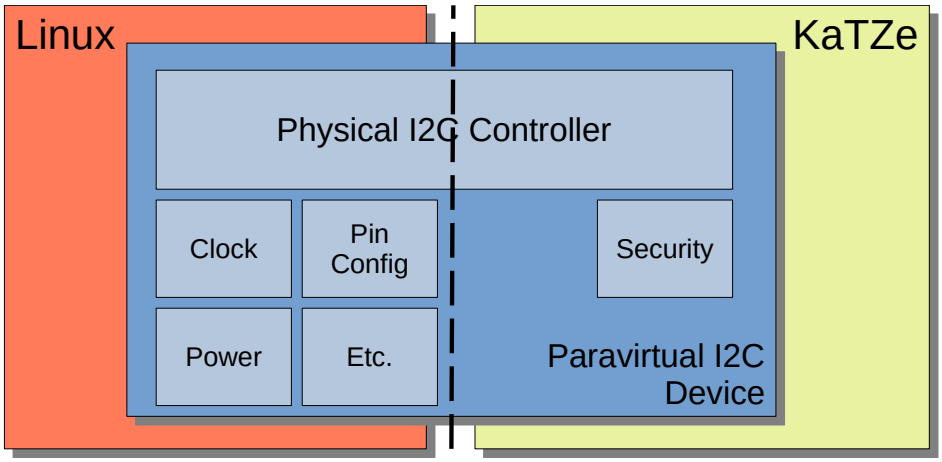

Figure 19: The functionality of a single "composite" paravirtualized device is provided by several discrete hardware devices.

5.3.1 The "Composite" Devices of the Paravirtual Framework

A direct result of paravirtualization is the logical reduction of hardware complexity from the paravirtual client's perspective. When a framework like virtio exposes a block controller it exposes a "generic" version while the corresponding device underneath is a specific block controller from a specific vendor, which is accessed over a specific bus. This device may require configuration "inputs" from other parts of the system, including the IOMMU and interrupt controller, as shown in Figure 19.

The graph of device functionality dependencies takes on three forms, based on perspective: the host system, the paravirtual framework, and the client system. They are depicted in Figure 20. The host system perspective is typical, fully considering each device's interconnections, requiring a complex graph dependency system in Linux to resolve, including asynchronous initialization callbacks [55]. By comparison our paravirtual framework "trims" the graph, i.e. it abstracts the block controller as the device plus its dependencies. Finally,

the client system perspective sees only the paravirtualized block controller, which may be a generic block controller or the actual device itself, as in the case of the RK3399's ISP.

From the paravirtual perspective, the trusted partition is the client and has the most *condensed* view of the underlying hardware. The client interacts with the paravirtual framework layer by making requests to the paravirtual host, which is in the untrusted partition. In particular, the half of the split driver that resides in Linux and controls the device is the paravirtual host, which leverages the existing Linux device system to provide the device service.

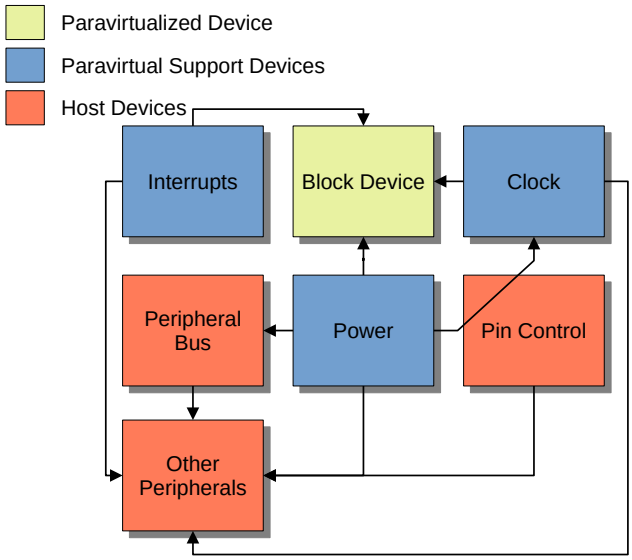

Figure 20: Our paravirtual framework organizes the system's devices into three perspectives, or "layers" of increasing granularity or abstractness.

5.3.2 Containing Platform Complexity

The split driver model also works to contain the complexity of the underlying SoC platform in IoT sensing environments. These systems are almost universally SoCs, with vendors providing massively different packages, capabilities, and hardware revision versions; even within a single vendor's SoC lines there will be variations. Device vendors *do* have some incentive to provide drivers to mainstream kernels like Linux or FreeBSD, but not much incentive or scope to do it securely. While the review process of open-source projects like Linux puts in a floor of quality for contributions it is not sufficient to provide security guarantees. Fortunately, the delegation allows us to bypass this concern and use these drivers in-place

in an untrusted manner.

By limiting the trusted OS drivers to only the sensors themselves we constrain the complexity of the SoC platforms. This can be seen by example in Figure 21. Assume that the trusted partition for a given system has been specified, configured, audited, and deployed. It will contain some paravirtual device drivers, such as a camera driver or the ISP driver. This trusted partition is typically packaged into an image and signed in some way that enables attestation, discussed later in Section 6.1. This image is only meant to change infrequently and thus is often bundled as part of the system's firmware image.

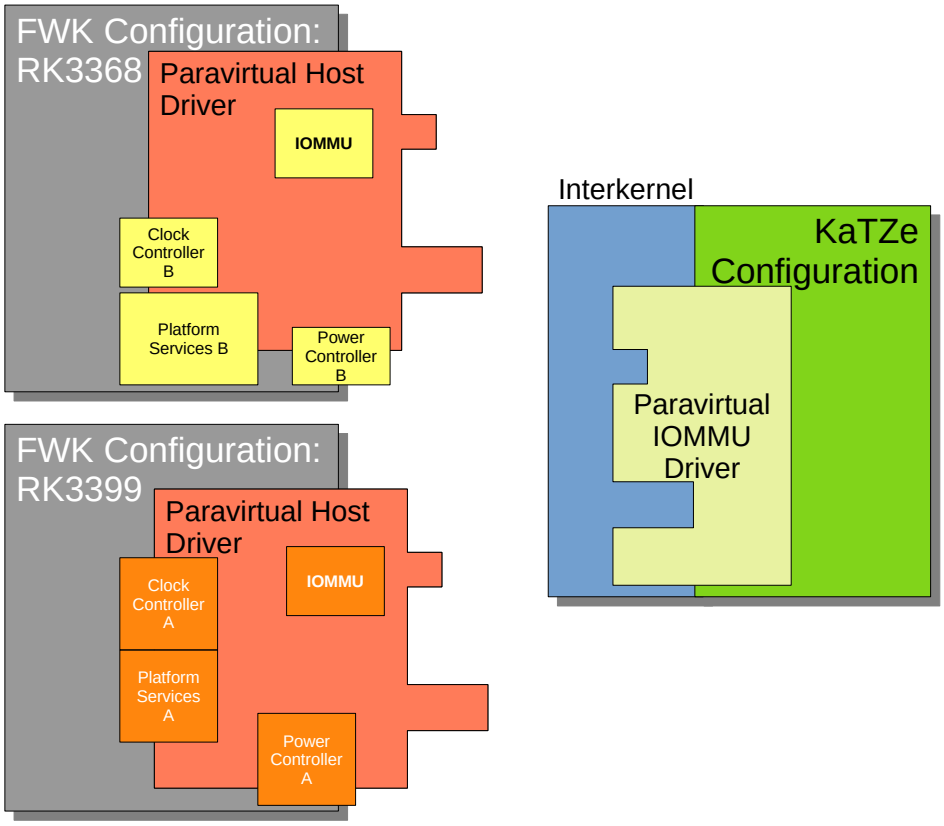

Figure 21: Paravirtual drivers expose a "key" to the untrusted partition, which various underlying platforms can fit to provide services to the TEE.

The *trusted* portion of the device driver cannot be changed without a re-evaluation of the whole trusted partition. However, a paravirtualized trusted partition image allows a degree of flexibility to the underlying platform. Consider by example the other systems produced by Rockchips than the RK3399 platform. It is known from the Linux driver [112] that this same IOMMU block has been used in other Rockchip SoCs, including the RK3368, the RK3036, the RK3328, and others. While the device itself functions the same, its device dependencies

differ. Without paravirtualization the overall driver configuration of the working kernel must change. But with the split, paravirtualized driver, a trusted partition image can freely change certain underlying devices without changing the drivers contained in the trusted partition image.

5.4 Implementation of Split Drivers in KaTZe

We realize our split driver design in KaTZe with the help of TrustZone's flexible capabilities. Guo and others [31, 13] reinforce that a key issue with the TrustZone ecosystem is the lack of drivers, and we can speculate that the reasons may be that there is too little interest, the OS and platform are too hard to write for, or the devices are too complex to port drivers. Thanks to configurable controllers attached to most busses in TrustZone platforms, including the DMA controllers, interrupt controllers, and main bus controllers, it is possible to discriminate accesses based on the security state of the accessor, and even assigning finer-grained read-write permissions to these accesses.

5.4.1 Privacy and Security Considerations

In our motivating use case, the Privacy Backplane, we accept that protecting availability is impossible, considering that on all nodes in the system there is a co-extant, potentially adversarial kernel that has kernel-level but non-secure access to the same underlying hardware. In the backplane context, someone whose data has been collected is primarily concerned with the *misuse* of their data–they want their privacy policy enforced so that frames showing their faces must be anonymized before use, for example. In the naive case, the infrastructure owners, i.e. the potential adversary, also have an interest in data integrity, as ruining sensor data makes it inaccessible for both the trusted backplane and any untrusted clients that interact with it.

Though we must concede availability, we contend that we can retain confidentiality and integrity. Consider the example of the D-PHY bus found on our RK3399 system, shown in

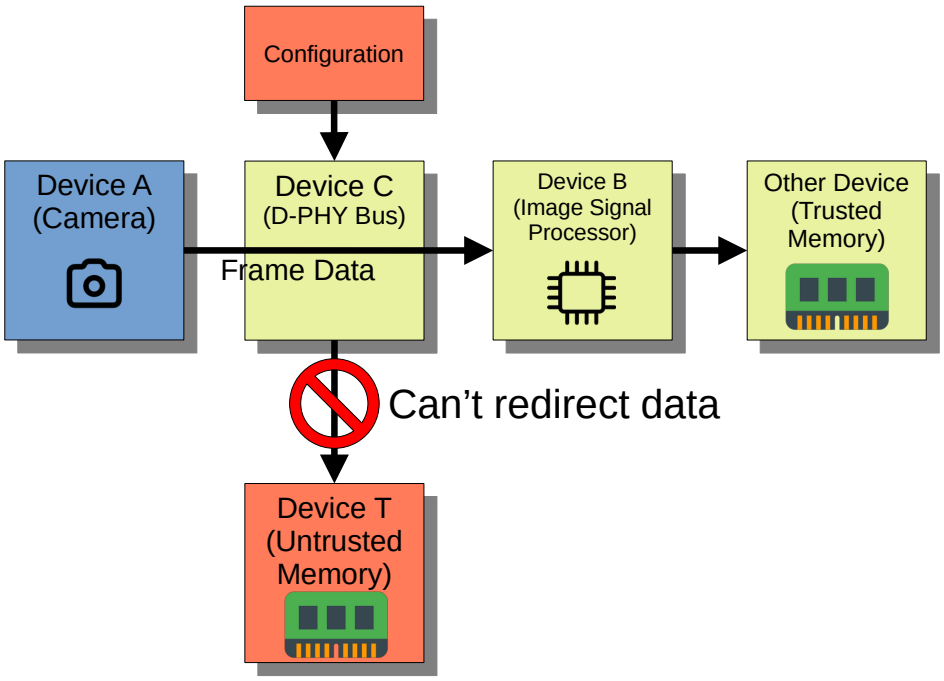

Figure 22: If input A linked to output B are trusted and a midpoint C cannot be configured to output to an adversary, then C's *data* is trusted even when *control* is not.

Figure 22. The bus functions as a continuous stream of data, and configuration ultimately determines the interval at which an arbitrary "frame end" signal is delivered, indicating to the output device to consume a complete frame. Misconfiguration of this device results in garbled inputs to the downstream devices. However, because of physical limitations of the system it is impossible to redirect data on the bus itself–it always flows into the ISP block. Thus we consider that the control of the bus is untrusted but the data on the bus is trusted. As it turns out, this pattern is found in many devices on SoC systems and exploiting this pattern allows substantial reduction of trusted partition driver code by re-using existing driver code in the untrusted partition without compromising data security.

5.4.2 Re-Using Existing Linux Configurations

In our implementation we leverage the fact that Linux automatically configures platform resources for nodes in its device tree, a system configuration format that describes resources and their interdependence [58]. A device tree describes a device's location in memory, what

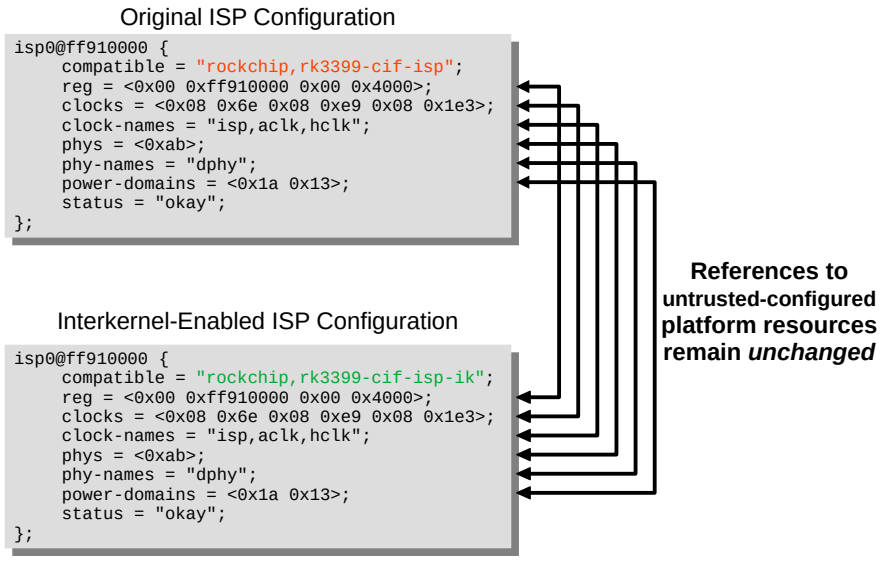

Figure 23: By changing only the selected Linux driver, our implementation re-uses valid Linux configurations to enable a trusted paravirtual device.

driver to use, and the platform resources it requires, such as clocks, I/O pin configurations, and power domains. In our driver framework we benefit from this auto-configuration by substituting the existing driver for a paravirtual one. This is shown in Figure 23. This slotting technique provides a straightforward pathway for augmenting existing Linux drivers incrementally, as the existing driver code-bases can be augmented with the interkernel capability by hooking into a central dispatch system similar to interrupt handlers.

To fully support the RKISP1 ISP requires configuring several other components: the D-PHY bus, its power domain, a clock source, the access control of its MMIO range, and the input camera itself. Access control is a TrustZone configuration and is done by the trusted partition. The other configurations are part of the platform and controlled by Linux. We leverage Linux's device tree functionality to auto-configure some of these devices, and for the others we trade power-efficiency for simplicity and mark these platform devices as "always-on," which prevents the Linux power management system from disabling them. We further elide the power management system by choosing to configure the drivers at system initialization time, guaranteeing that the devices such as the bus, power, and clock will be in a known-good state when the paravirtual framework configures peripherals that depend on these platform devices.

These short descriptions illustrate how the separation of the control and data planes facilitates driver decomposition, allowing re-use of existing drivers by integrating with the Linux device infrastructure. By securing the data plane directly by implementing drivers in KaTZe and leveraging the untrusted Linux kernel for the non-secure control plane, we demonstrate a general model for secure sensor access that can apply to other sensors as well.

5.4.3 Device Limitations: Memory Access Granularity

Due to the limitations of TrustZone hardware, not all devices can be decomposed along their data-control axis. Current implementations lack sufficient resolution to separately choose the security state of all devices. Specifically, the TrustZone controller on the RK3399 SoC secures memory in 2MB chunks, while several SoC sub-devices are mapped into memory with more than one device per 2MB-region. To achieve per-device security states, hardware must either: map devices more sparsely inside the address space, or improve the granularity of security regions. This still limits device decomposition to *per-device*, when optimally it would be *intra-device* for most flexibility. Devices such as UARTs where data registers are intermingled with control registers in the same memory page cannot be secured without byte granularity. With less granularity UART-like devices can be owned by the trusted OS, but not paravirtualized. Capability systems like the recent CHERI[111] permit byte-level granularity on access capabilities, which can supplement or perhaps replace the existing TrustZone controller depending on the capability implementation.

The problem of granularity when securing the data plane is reduced if a device is itself capable of writing to memory; though a UART outputs data by being read from at certain registers, some devices on modern SoCs are "push capable," such as the RK3399 ISP. One of its output options is a scatter-gather model where memory addresses are supplied to the device and frames are written to those addresses by the device asynchronously to the CPU. However, this functionality still does not permit intra-device decomposition and such devices still cannot be easily paravirtualized.

5.5 Evaluation

To demonstrate the viability of the paravirtual, split-driver approach we tested both the quantitative performance as well as the qualitative advantage of the system. We improve our previous image recognition pipeline by paravirtualizing necessary devices. We benchmarked the SOD image recognition, described in Section 4.5.2, on captured frames on Linux and KaTZe and measured the sampling rate of the HTU21D sensor under both the ported driver as well as its paravirtual counterpart. Our image recognition pipeline employs paravirtual driver components in non-performance critical code paths, demonstrating both the feasibility of direct I/O in the KaTZe kernel as well as the utility of the paravirtual code; in that benchmark, control of the camera is achieved using a paravirtualized I2C driver, as well as the supporting platform hardware, including the clocks, power supplies, and I/O pin configurations needed for the I2C busses, camera busses, and the ISP itself. To measure qualitative advantages we analyze the TCB reductions achieved by using paravirtual drivers, and describe the reusability of the paravirtual drivers as it applies to other systems using the same underlying device. The underlying hardware design that allows for paravirtualization is not specific to the RK3399 SoC and is applicable to other SoCs as well. The set of devices we chose, namely: core platform devices (power, pin controls, clock), peripheral devices (HTU21D sensor), integrated SoC devices (RKISP1 ISP), and high-speed peripherals (IMX214 camera) and associated busses (D-PHY) span a wide variety of relevant devices on these platforms and demonstrate the approach's generality.

5.5.1 HTU21D: Paravirtual vs Ported

To show the performance overheads inherent in paravirtualizing a driver itself, we created a paravirtualized version of the I2C controller driver and reconfigured our existing, ported HTU21D driver to use the paravirtual backend. This benchmark is constructed and is not secure in the context of the HTU21D sensor itself, but could be acceptable in other contexts when only the control plane for a device is exposed over the I2C bus, such as the IMX214 camera. The principal difference between these two systems is the I2C controller; in the

paravirtualized system, the Kitten kernel driver is paravirtualized, sending commands and data to a corresponding Linux kernel host driver, which forwards the commands to the physical I2C controller. In the native system, this work happens entirely in the Kitten kernel without the involvement of the Linux kernel. The results are shown in Figure 24. The native driver on average takes 54ms (18 samples/second) to acquire a sample from the device as compared to the paravirtual driver's 82ms (12 samples/second). For comparison, we performed two extra experiments, the latency for a simple ping-pong, as well as a ping-pong which performs a copy of dummy data. At both over an equivalent 2400 samples/second (˜410 microseconds per sample), it is clear that the inter-core communication is not the bottleneck. While a control plane for a camera is not performance-critical, this benchmark demonstrates that paravirtualizing a device can be acceptable, especially for a low-speed, low-frequency bus such as I2C where sampling rates for any attached device will be low. Meanwhile, this small performance trade-off allows a nearly complete delegation of complex, error-prone hardware handling code to the untrusted kernel, resulting in an improved TCB for the secure kernel.

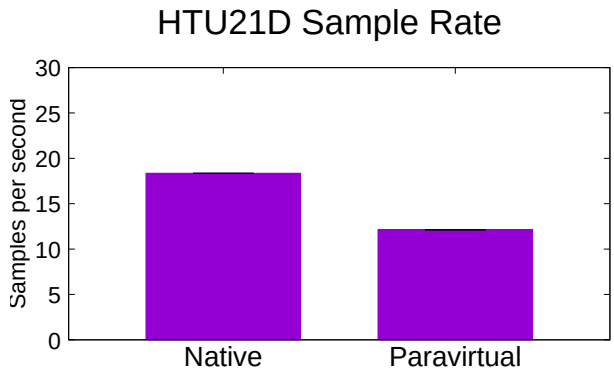

Figure 24: The HTU21D paravirtualized driver achieves a reasonable fraction, 0.626x the performance of a native driver equivalent.

5.5.2 TCB Reduction

Part of the motivation for using paravirtualization is the de-privileging of driver code. To investigate how much unneeded code we can move out of the trusted partition, we look at the drivers involved in a few key areas. We consider platform code separately from driver code, and we look at both the drivers needed to run the HTU21D sensor as well

as those needed to run the camera pipeline. Platform code offers the greatest advantages, as the trusted partition does not have a *requirement* to interact with most platform code, and thus the Kitten kernel does not need to even be aware of this functionality. However, for other concerns, such as camera management, a simplified paravirtual driver still masks the entirety of the hardware-specific code from the trusted partition, presenting these as generic ops like "start streaming" or "set exposure." Our tabulation of the driver code left out of the Kitten kernel is shown in Table 3. It is further worth noting that, given time and functionality constraints, not every driver has been fully optimized for size when porting to the Kitten kernel, such as the ISP block which contains inactive code that provides unneeded capability, such as image effects. We also did not attempt to tabulate the full driver infrastructure weight that each of these individual drivers is plugged into. This leaves out of the Linux tabulation a not-insignificant amount of code; Linux's I2C framework alone is 4870 LOC. Regardless, a full investigation and analysis of driver infrastructure is well beyond the scope of evaluating this paravirtualization technique. This analysis demonstrates that paravirtualization as a technique provides substantial reductions, providing the most benefit for platform-wide device. Moreover, the Kitten kernel's comparative simplicity allows even complex device drivers to be trimmed somewhat of the Linux device driver boilerplate.

5.5.3 Conclusion

This evaluation demonstrates Research Insight 4, that substantial gains in capability can be achieved by aggressively applying a multi-kernel framework which allows the trusted partition to leverage the existing untrusted partition. In contrast to the prior frameworks that are of the "enclave" model, we leverage robust TEE hardware and rely on it to protect the trusted partition, which allows close interaction between the partitions. Indeed, this integration is pivotal to our system, suggesting that secure, capable systems ought to rely on this multi-kernel integration more.

Table 3: Paravirtualizing the platform drivers results in an average reduction of ~57% per device, or an overall LOC reduction of 53%.

	Original LOC	Paravirtual LOC	Percentage Reduction
Platform Management			
Clock Management	1346	0[a]	100%
GPIO Pin Control	2680	0[a]	100%
Power Management	891	0[a]	100%
HTU21D Driver Stack			
I2C Controller	837	735	12.2%
HTU21D Sensor	200	117	41.5%
IMX214 Camera/RK3399 ISP Driver Stack			
RKISP1 ISP	4699	3909	16.8%
Rockchip IOMMU	975	1082[b]	-11%
IMX214 Camera	888	0	100%[c]

[a]By exploiting default, always-on configurations, the Kitten kernel can be completely unaware of this functionality.

[b]A good example of where Linux's large inventory of utility and convenience functions allows code re-use across drivers.

[c]The camera's dataflow cannot be retargeted, but certain controls can, in theory, compromise security, such as adjusting exposure to subvert facial recognition.

6.0 Discussion

The KaTZe system provides broad support for an important application class in the edge/IoT space, along with secure I/O and a driver model that leverages the existence of an untrusted full-weight kernel. However, this forms only the system layer of a distributed trusted application like the Privacy Backplane as described in the Section 1.1. An important but out-of-scope component of such an application is the attestation protocol, which we discuss to illustrate KaTZe's architectural compatibility.

6.1 Support for Attestation Mechanisms

The goal of attestation is verifying that a program is in a known, trusted state and is generally classed as two problems based on the perspective of the observer: local and remote. For distributed trusted applications some scheme for remote attestation is necessary to establish trust, while local attestation suffices for the trust envelope of a single machine. Parno has previously given a thorough description of the attestation problem [77]. Briefly, to achieve remote attestation, or *trusted boot*, the system must accumulate "measurements" or "evidence" that describes the state of programs that have run. This process is largely orthogonal to the architecture of the system software so long as it collects these measurements. Another consideration is how to store the measurements in a secure way, which can be accomplished either by storing them in a local TPM, or employing a certificate chain to secure them. At any stage of the boot process these measurements can be compared with stored, previously-computed values to determine if software has been tampered with and so prevent execution, making the boot process secure. Naturally a general-purpose OS is capable of interacting with a TPM and recording hashes of programs, provided drivers are available. Resultingly there are many TEEs that are compatible with local attestation, and as a general-purpose OS within the trusted partition, KaTZe is capable of integrating into existing local attestation/trusted boot chains.

For distributed trust, however, the situation becomes more complicated. Remote attestation is the process of using the measurements or evidences to prove local state to a remote verifier. To achieve this, a cryptographic identity must be present on the local attestor that is known to the remote verifier, established out of band. To prevent tampering, this identity is permanently bound to the system using one time-programmable memory. Again, a general-purpose OS with drivers is sufficient to achieve this. There must exist a secure channel between the remote verifier and local attestor and with techniques like TLS/SSL a "virtual" secure channel can be established atop existing untrusted channels.

Remote attestation becomes substantially more complex when repeated attestation is involved. Once an application is running, taking measurements becomes more difficult, as measurements must capture state that can affect the good behavior of the application, but the known-good values of these measurements must be available to the verifier. Thus a tension between the size of the allowable state space and the difficulty to verify that state space exists. Formally-verified kernels such as seL4 [40] provide proofs of correctness and security that can simplify the remote attestation process, but they do not solve the problem of attesting applications themselves, such as a database application or a machine learning model. Current research to provide remote attestation of applications is with a sandboxed runtime, such as the WaTZ sandboxed WebAssembly engine [68]. KaTZe can support the WaTZ protocol, providing the same remote attestation support for applications. While KaTZe is not proven correct, deploying an application-layer attestation mechanism like WaTZ provides a "secure foundation" upon which a distributed, trusted application can be built; a secure system image containing the KaTZe OS and WaTZ runtime would be audited, verified, and then deployed. Though outside the scope of this work, remote attestation is a critical component of a distributed trusted application that the KaTZe system is architecturally compatible with.

6.2 Conclusion and Further Work

In this thesis a new capability in trusted computing was demonstrated. To accommodate the difficult combination of rich application and I/O access, which has traditionally been tackled separately, we identified three key properties that such a system would need:

- Supports existing applications currently in-use, without requiring re-design for a specific security framework
- Support a direct, secure path to devices and expose them to applications
- Do these things without bloating the TCB of the secure kernel needlessly.

In the KaTZe system and our research, we have described such a system. In Chapter 3 we described how the LWK concept provides broad POSIX capability, allowing the majority of existing applications that are programmed with mainstream OS abstractions in mind to run, without modification. Indeed, development of KaTZe programs can be done on Linux and those binaries tested on a Linux platform. Further, when deployed to the KaTZe LWK TEE kernel they remain performant for the task at hand. In Chapter 4 we explained that despite the difficult situation of I/O devices in the Linux kernel on current edge/IoT platforms, a LWK such as KaTZe can support relevant devices, even complex ones such as cameras and image processors, while simultaneously trimming the fat from those drivers. In Chapter 5 we describe how paravirtualization can be applied to securely delegate large portions of code, specifically that code which is needed to support common hardware platform devices. We further apply this technique by recognizing that even our sensors themselves may have security non-critical components and so we can delegate those control surfaces away, further reducing the TCB of the secure kernel.

Bibliography

[1] Tigist Abera, N. Asokan, Lucas Davi, Farinaz Koushanfar, Andrew J. Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. Invited: Things, trouble, trust: On building trust in iot systems. *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2016.

[2] Abdullah Al-Boghdady, Khaled Wassif, and Mohammad El-Ramly. The presence, trends, and causes of security vulnerabilities in operating systems of iot's low-end devices. *Sensors*, 21(7), 2021.

[3] AMD. *AMD I/O Virtualization Technology (IOMMU) Specification*, October 2023. `https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/specifications/48882_IOMMU.pdf`.

[4] Android Open Source Project. Trusty TEE, 2023.

[5] ARM. *Arm Architecture Reference Manual for A-profile architecture*, June 2024.

[6] ARM. *ARM CoreLink TZC-400 TrustZone Address Space Controller Technical Reference Manual*, June 2024. `https://developer.arm.com/documentation/ddi0504/latest/`.

[7] ARM. Learn the architecture - generic interrupt controller v3 and v4 - overview, 2024.

[8] ARM. Mali-c55, 2024.

[9] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, André Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, David Goltzsche, D. Eyers, Rüdiger Kapitza, Peter R. Pietzuch, and Christof Fetzer. Scone: Secure linux containers with intel sgx. In *USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[10] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Cure: A security architecture with customizable and resilient enclaves. In *USENIX Security Symposium*, 2020.

[11] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *Operating Systems Review*, 2003.

[12] Andrew Baumann, Marcus Peinado, and Galen C. Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)*, 33:1 – 26, 2014.

[13] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Sanctuary: Arming trustzone with user-space enclaves. *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.

[14] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: Sgx cache attacks are practical. *ArXiv*, abs/1702.07521, 2017.

[15] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. On-demand container loading in aws lambda. In *USENIX Annual Technical Conference*, 2023.

[16] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1416–1432, 2020.

[17] Cesanta. Mongoose - an embedded Web Server, MQTT and Websocket library, 2024.

[18] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-sgx: A practical library os for unmodified applications on sgx. In *USENIX Annual Technical Conference*, 2017.

[19] Collabora. `https://github.com/torvalds/linux/tree/master/drivers/media/platform/rockchip/rkisp1`, June 2024.

[20] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. Secure processors part ii: Intel sgx security analysis and mit sanctum architecture. *Found. Trends Electron. Des. Autom.*, 11:249–361, 2017.

[21] Karim M. El Defrawy, Norrathep Rattanavipanon, and Gene Tsudik. Hydra: hybrid design for remote attestation (using a formally verified microkernel). *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2017.

[22] Saba Eskandarian, Jonathan Cogan, Sawyer Birnbaum, Peh Chang Wei Brandon, Dillon Franke, Forest Fraser, Gaspar Garcia, Eric Gong, Hung T. Nguyen, Taresh K. Sethi, Vishal Subbiah, Michael Backes, Giancarlo Pellegrino, and Dan Boneh. Fidelius: Protecting user secrets from compromised browsers. *2019 IEEE Symposium on Security and Privacy (SP)*, pages 264–280, 2018.

[23] Erhu Feng, X. Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Scalable memory protection in the penglai enclave. In *USENIX Symposium on Operating Systems Design and Implementation*, 2021.

[24] Andreas Fitzek, Florian Achleitner, Johannes Winter, and Daniel M. Hein. The andix research os — arm trustzone meets industrial control systems security. *2015 IEEE 13th International Conference on Industrial Informatics (INDIN)*, pages 88–93, 2015.

[25] Balazs Gerofi, Yutaka Ishikawa, Rolf Riesen, Robert W. Wisniewski, Yoonho Park, and Bryan S. Rosenburg. A multi-kernel survey for high-performance computing. *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*, 2016.

[26] GlobalPlatform. TEE Client API Specification, 2023.

[27] Google. Widevine, 2024.

[28] Nicholas Gordon and John R. Lange. Lifting and dropping vms to dynamically transition between time- and space-sharing for large-scale hpc systems. *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, 2022.

[29] Nicholas Gordon, Kevin T. Pedretti, and John R. Lange. Porting the kitten lightweight kernel operating system to risc-v. *2022 IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, pages 1–7, 2022.

[30] Nicholas Gregory and Harini Kannan. Using undocumented hardware performance counters to detect spectre-style attacks. In *Conference on Applied Machine Learning in Information Security (CAMLIS)*, 2021.

[31] Liwei Guo and Felix Xiaozhu Lin. Minimum viable device drivers for arm trustzone. *Proceedings of the Seventeenth European Conference on Computer Systems*, 2021.

[32] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017.

[33] Wei Huang, Vasily Rudchenko, He Shuang, Zhen Huang, and David Lie. Pearl-tee: Supporting untrusted applications in trustzone. *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, 2018.

[34] Guerney D. H. Hunt, Ram Pai, Michael V. Le, Hani Jamjoom, Sukadev Bhattiprolu, Richard H. Boivie, Laurent Dufour, Brad Frey, Mohit Kapur, Kenneth A. Goldman, Ryan Grimm, Janani Janakirman, John M. Ludden, Paul Mackerras, Cathy May, Elaine R. Palmer, Bharata Bhasker Rao, Lawrence Roy, William A. Starke, Jeffrey Stuecheli, Enriquillo Valdez, and Wendel Voigt. Confidential computing for openpower. *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021.

[35] Intel. Intel Software Guard Extensions, 2023.

[36] Intel. *Intel Virtualization Technology for Directed I/O*, March 2023. `http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf`.

[37] Blake Ives, Kathy L. Cossick, and Dennis A. Adams. Amazon go: Disrupting retail? *Journal of Information Technology Teaching Cases*, 9:12 – 2, 2019.

[38] Dong Ji, Qianying Zhang, Shijun Zhao, Zhiping Shi, and Yong Guan. Microtee: Designing tee os based on the microkernel architecture. *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 26–33, 2019.

[39] Matthieu Jimenez, Mike Papadakis, and Yves Le Traon. An empirical analysis of vulnerabilities in openssl and the linux kernel. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 105–112, 2016.

[40] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David A. Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In *Symposium on Operating Systems Principles*, 2009.

[41] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual International Cryptology Conference*, 1999.

[42] Hsuan-Chi Kuo, Jianyan Chen, Sibin Mohan, and Tianyin Xu. Set the configuration for the heart of the os. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4:1 – 27, 2020.

[43] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. A linux in unikernel clothing. *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020.

[44] Argonne National Laboratories. Argonne national laboratories benchmarks repo.

[45] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. Empirical analysis of the relationship between cc and sloc in a large corpus of java methods and c functions. *Journal of Software: Evolution and Process*, 28:589 – 618, 2016.

[46] John Lange, Peter A. Dinda, Robert Dick, Friedrich Doku, Elena Fabian, Nick Gordon, Peizhi Liu, Michael Polinski, Madhav Suresh, Carson Surmeier, and Nick Wanninger. A case for a user-centered distributed privacy backplane for the internet of things. Technical report, Northwestern University, 2023.

[47] John R. Lange, Nicholas Gordon, and Brian L. Gaines. Low overhead security isolation using lightweight kernels and tees. *2021 SC Workshops Supplementary Proceedings (SCWS)*, pages 42–49, 2021.

[48] Stefan Lankes, Simon Pickartz, and Jens Breitbart. HermitCore: a Unikernel for Extreme Scale Computing. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*, 2016.

[49] Michael Larabel. Linux to drop support for 15 year old, never-shipped intel "carillo ranch", 2023.

[50] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Xiaodong Song. Keystone: an open framework for architecting trusted execution environments. *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020.

[51] Joshua LeVasseur and Volkmar Uhlig. A sledgehammer approach to reuse of legacy device drivers. In *EW 11*, 2004.

[52] Congmiao Li and Jean-Luc Gaudiot. Detecting malicious attacks exploiting hardware vulnerabilities using performance counters. *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, 1:588–597, 2019.

[53] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. Cipherleaks: Breaking constant-time cryptography on amd sev via the ciphertext side channel. In *USENIX Security Symposium*, 2021.

[54] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, D. Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter R. Pietzuch. Glamdring: Automatic application partitioning for intel sgx. In *USENIX Annual Technical Conference*, 2017.

[55] Device drivers - the linux kernel documentation. `https://www.kernel.org/doc/html/latest/driver-api/driver-model/driver.html`, June 2024.

[56] Littlekernel. Github - littlekernel/lk, 2023.

[57] Jiuxing Liu, Wei Huang, Bülent Abali, and Dhabaleswar Kumar Panda. High performance vmm-bypass i/o in virtual machines. In *USENIX Annual Technical Conference, General Track*, 2006.

[58] Linaro Ltd. Devicetree specifications, 2024.

[59] John Madieu. *Mastering Linux Device Driver Development: Write custom device drivers to support computer peripherals in Linux operating systems.* Packt Publishing, 2021.

[60] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering intel last-level cache complex addressing using performance counters. In *International Symposium on Recent Advances in Intrusion Detection*, 2015.

[61] John D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[62] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: an execution infrastructure for tcb minimization. In *European Conference on Computer Systems*, 2008.

[63] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Arvind Seshadri. How low can you go?: recommendations for hardware-supported minimal tcb code execution. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[64] Jonathan M. McCune, Adrian Perrig, and Michael K. Reiter. Safe passage for passwords and other sensitive data. In *Network and Distributed System Security Symposium*, 2009.

[65] Brian McGillion, Tanel Dettenborn, Thomas Nyman, and N. Asokan. Open-tee – an open virtual trusted execution environment. *2015 IEEE Trustcom/BigDataSE/ISPA*, 1:400–407, 2015.

[66] Rebecca T. Mercuri and Peter G. Neumann. Security by obscurity. *Commun. ACM*, 46:160, 2003.

[67] Fan Mo, Hamed Haddadi, Kleomenis Katevas, Eduard Marin, Diego Perino, and Nicolas Kourtellis. Ppfl: privacy-preserving federated learning with trusted execution environments. *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, 2021.

[68] James Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. Watz: A trusted webassembly runtime environment with remote attestation for trustzone. *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*, pages 1177–1189, 2022.

[69] Cong T. Nguyen, Yuris Mulya Saputra, Nguyen Van Huynh, Ngoc-Tan Nguyen, Tran Viet Khoa, Bui Minh Tuan, Diep N. Nguyen, Dinh Thai Hoang, Thang Xuan Vu, Eryk Dutkiewicz, Symeon Chatzinotas, and Björn E. Ottersten. A comprehensive survey of enabling and emerging technologies for social distancing—part i: Fundamentals and enabling technologies. *Ieee Access*, 8:153479 – 153507, 2020.

[70] National Institute of Standards and Technology. Post-processing audit tools and techniques. Technical report, U.S. Department of Commerce, 1977.

[71] A. Oliveira, José Martins, Jorge Cabral, Adriano Tavares, and Sandro Pinto. Tz- virtio: Enabling standardized inter-partition communication in a trustzone-assisted hypervisor. *2018 IEEE 27th International Symposium on Industrial Electronics (ISIE)*, pages 708–713, 2018.

[72] OP-TEE. TLS support in OPTEE, 2023.

[73] Oracle. Introduction to virtio. `https://blogs.oracle.com/linux/post/introduction-to-virtio`, 2024.

[74] Jiannan Ouyang, Brian Kocoloski, John R. Lange, and Kevin Pedretti. Achieving Performance Isolation with Lightweight Co-Kernels. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, 2015.

[75] Jiannan Ouyang, Brian Kocoloski, John R. Lange, and Kevin T. Pedretti. Achieving performance isolation with lightweight co-kernels. *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, 2015.

[76] European Parliament. General data protection regulation.

[77] Bryan Parno, Jonathan M. McCune, and Adrian Perrig. Bootstrapping trust in modern computers. In *Springer Briefs in Computer Science*, 2011.

[78] Leonardo Passos, Rodrigo Queiroz, Mukelabai Mukelabai, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Jesus Alejandro Padilla. A study of feature scattering in the linux kernel. *IEEE Transactions on Software Engineering*, 47(1):146–164, 2021.

[79] Kevin Pedretti. Kitten: A Lightweight Operating System for Ultrascale Supercomputers. *Sandia Lab*, 2011.

[80] Roberto Di Pietro, Flavio Lombardi, and Antonio Villani. Cuda leaks. *ACM Transactions on Embedded Computing Systems (TECS)*, 15:1 – 25, 2013.

[81] Sandro Pinto and Nuno Santos. Demystifying arm trustzone. *ACM Computing Surveys (CSUR)*, 51:1 – 36, 2019.

[82] PixLab. SOD - An Embedded, Modern Computer Vision and Machine Learning Library, 2023.

[83] Davide Quarta, Michele Ianni, Aravind Machiry, Yanick Fratantonio, Eric Gustafson, Davide Balzarotti, Martina Lindorfer, Giovanni Vigna, and Christopher Kruegel. Tarnhelm: Isolated, transparent & confidential execution of arbitrary code in arm's trustzone. *Proceedings of the 2021 Research on offensive and defensive techniques in the Context of Man At The End (MATE) Attacks*, 2021.

[84] Himanshu Raj, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Löser, Dennis Mattoon, Magnus Nyström, David Robinson, Rob Spiger, Stefan Thom, and David Wooten. ftpm: A software-only implementation of a tpm chip. In *USENIX Security Symposium*, 2016.

[85] Rolf Riesen, Arthur B. Maccabe, Balazs Gerofi, David N. Lombard, Jack Lange, Kevin T. Pedretti, Kurt B. Ferreira, Mike Lang, Pardo Keppel, Robert W. Wisniewski, Ron Brightwell, Todd Inglett, Yoonho Park, and Yutaka Ishikawa. What is a lightweight kernel? *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*, 2015.

[86] Rockchips. *Rockchip RK3399 TRM*, 2017. `https://opensource.rock-chips.com/images/e/ee/Rockchip_RK3399TRM_V1.4_Part1-20170408.pdf`.

[87] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Oper. Syst. Rev.*, 42:95–103, 2008.

[88] Leonid Ryzhyk, Yanjin Zhu, and Gernot Heiser. The case for active device drivers. In *Asia Pacific Workshop on Systems*, 2010.

[89] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Using arm trustzone to build a trusted language runtime for mobile applications. *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, 2014.

[90] Moritz Schneider, Ramya Jayaram Masti, Shweta Shinde, Srdjan Capkun, and Ronald Perez. Sok: Hardware-supported trusted execution environments. *ArXiv*, abs/2205.12742, 2022.

[91] Ioannis Sfyrakis and Thomas Gross. Uniguard: Protecting unikernels using intel sgx. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 99–105, 2018.

[92] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, and Yubin Xia. Occlum: Secure and efficient multitasking inside a single enclave of intel sgx. *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[93] Simon Shillaker and Peter R. Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *USENIX Annual Technical Conference*, 2020.

[94] Shweta Shinde, Dat Le Tien, Shruti Tople, and P. Saxena. Panoply: Low-tcb linux applications with sgx enclaves. In *Network and Distributed System Security Symposium*, 2017.

[95] Junaid Shuja, Abdullah Bin Gani, Kashif Bilal, Atta ur Rehman Khan, Sajjad Ahmad Madani, Samee Ullah Khan, and Albert Y. Zomaya. A survey of mobile device virtualization. *ACM Computing Surveys (CSUR)*, 49:1 – 36, 2016.

[96] SQLite Consortium. SQLite Home Page, 2024.

[97] Gookwon Edward Suh and Srinivas Devadas. Physical unclonable functions for device authentication and secret key generation. *2007 44th ACM/IEEE Design Automation Conference*, pages 9–14, 2007.

[98] TPC. TPC-H Decision Support Benchmark, 2024.

[99] TrustedFirmware. OP-TEE, 2023.

[100] TrustedFirwmare. TrustedFirmware-A (TF-A), 2023.

[101] TrustKernel. T6 - Secure OS and TEE, 2023.

[102] Trustonic. Cyber security technology located at the deepest level of the device, 2023.

[103] U-Boot. The U-Boot Documentation, 2023.

[104] Volodymyr Shymanskyy. wasm3: A fast WebAssembly interpreter and the most universal WASM runtime, 2024.

[105] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted execution environments on gpus. In *USENIX Symposium on Operating Systems Design and Implementation*, 2018.

[106] Xueyang Wang and Jerry Backer. Sigdrop: Signature-based rop detection using hardware performance counters. *ArXiv*, abs/1609.02667, 2016.

[107] Nico Weichbrodt, Anil Kurmus, Peter R. Pietzuch, and Rüdiger Kapitza. Asyncshock: Exploiting synchronisation bugs in intel sgx enclaves. In *European Symposium on Research in Computer Security*, 2016.

[108] Samuel Weiser and Mario Werner. Sgxio: Generic trusted i/o path for intel sgx. *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, 2017.

[109] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. Timber-v: Tag-isolated memory bringing fine-grained enclaves to risc-v. *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.

[110] David Wheeler. SLOCCOUNT.

[111] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert M. Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468, 2014.

[112] Simon Xue and Daniel Kurtz. rockchip-iommu.c driver. `https://github.com/torvalds/linux/blob/master/drivers/iommu/rockchip-iommu.c`, 2024.

[113] Peterson Yuhala, Pascal Felber, Valerio Schiavoni, and Alain Tchana. Plinius: Secure and persistent machine learning model training. *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 52–62, 2021.

[114] Irene Zhang, Jing Liu, Amanda Austin, Michael L. Roberts, and Anirudh Badam. I'm not dead yet!: The role of the operating system in a kernel-bypass era. *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2019.

[115] Zongwei Zhou, Miao Yu, and Virgil D. Gligor. Dancing with giants: Wimpy kernels for on-demand isolated i/o. *2014 IEEE Symposium on Security and Privacy*, pages 308–323, 2014.

[116] Jianping Zhu, Rui Hou, Xiaofeng Wang, Wenhao Wang, Jiangfeng Cao, Boyan Zhao, Zhongpu Wang, Yuhui Zhang, Jiameng Ying, Lixin Zhang, and Dan Meng. Enabling

rack-scale confidential computing using heterogeneous trusted execution environment. *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1450–1465, 2020.

[117] Marc Zyngier. Arm gicv3 linux kernel driver source code, June 2024.