2018 IEEE European Symposium on Security and Privacy

Have your Pi and Eat it Too: Practical Security on a Low-cost Ubiquitous Computing Platform

Amit Vasudevan and Sagar Chaki
amitvasudevan@acm.org, chaki@sei.cmu.edu
SEI, Carnegie Mellon University

***Abstract*—Robust security on a commodity low-cost and popular computing platform is a worthy goal for today's Internet of Things (IoT) and embedded ecosystems. We present the first practical security architecture on the Raspberry Pi (Pi), a ubiquitous and popular low-cost compute module. Our architecture and framework — called uberPi — focuses on three goals which are keys to achieving practical security: commodity compatibility (e.g., runs unmodified Raspbian/Debian Linux) and unfettered access to platform hardware, performance (avg. 2%–6% overhead), and low trusted computing base and complexity (modular 5544 SLoC). We present a full implementation followed by a comprehensive evaluation and lessons learned. We believe that our contributions and findings elevate the Pi into a next generation, secure, low-cost IoT embedded computing platform.**

***Index Terms*—Raspberry Pi Micro-Hypervisor Security Architecture, Trap-Inspect-Forward, Peripheral and Interrupt Partitioning, Uberguest, Uberapps.**

1. Introduction

Security in today's burgeoning Internet of Things (IoT) embedded platforms is of paramount importance given the rate of vulnerabilities [31], [32]. However, achieving *practical security* on interconnected embedded platforms is a challenge since it has to balance cost, performance, commodity compatibility (i.e., be able to run commodity unmodified off-the shelf OS and applications), and unfettered development practices (e.g., full access to platform functionality and choice of programming languages and tools) to foster wide industry and developer backing, in turn propelling rapid prototyping and shorter time to market.

While there have been several research proposals towards embedded system security [3], [5], [12], [17]–[21], [21], [22], [24], [26]–[28], [30], [40], [42], [43], [48], [50], [53], [54], they either employ ad hoc and/or closed platforms sacrificing commodity compatibility and affecting development practices [12], [21], [22], [24], [26]–[28], [30], [40], [43], [48], [53], target high-end platforms [19], [20], [50] or propose costly add-on modules that only provide a constrained execution environment [17], [21], [42], [54] (cf. related work; §9).

To address these limitations, we asked ourselves two questions: **(Q1)** *what is the lowest-cost, off-the-shelf, highly popular IoT/embedded development platform available today; and* **(Q2)** *what security properties can we harness from it using a small trusted computing base (TCB), while remaining performant and embracing commodity compatibility and unfettered development practices?*

The answer to **Q1** is a credit-card size, sub $35 computing platform — the Raspberry Pi (Pi). The Pi has sold over several million units since its inception in 2012, and is one of the most popular development and prototyping platform today in the IoT/embedded space [1]. It runs commodity Linux (Raspbian, Ubuntu, etc.) and Windows (IoT core) OSes, and enjoys a huge developer support and industry adoption (IBM, Microsoft, Google). Last, but not least, Broadcom, which supplies the base Pi hardware, continues to provide incremental hardware improvements over 3 generations of the Pi. The answer to **Q2**, and our main contribution, is uberPi (uPi), a micro-hypervisor (μHV) based system security architecture and framework for the Pi.

We begin by presenting a hardware platform level architecture of the Pi, and discuss the interplay between relevant hardware platform primitives in the context of system security (§2). While existing documents describe parts of the Pi hardware solely from a programming and OS porting perspective, to our knowledge this is the first holistic description of the Pi platform architecture with a primary focus on system security.

Next, informed by the security oriented platform architecture of the Pi, we present the uPi architecture (§4). uPi embraces a μHV based system architecture supporting a single full-featured unmodified commodity guest OS (uberguest). We make this design decision to achieve commodity compatibility. Specifically, uPi leverages basic Pi platform hardware primitives to allow the uberguest direct access to all performance critical peripherals and interrupts. This model results in reduced μHV complexity (since all peripherals are directly controlled by the OS) and consequently TCB, as well as high performance (since guest peripheral interrupts do not trap to the hypervisor).

uPi uses a novel lightweight trap-inspect-forward (TIF) mechanism to selectively *trap* and *inspect* critical peripheral register accesses, before *forwarding* the access directly to

© 2018, Amit Vasudevan. Under license to IEEE.
DOI 10.1109/EuroSP.2018.00021

IEEE computer society

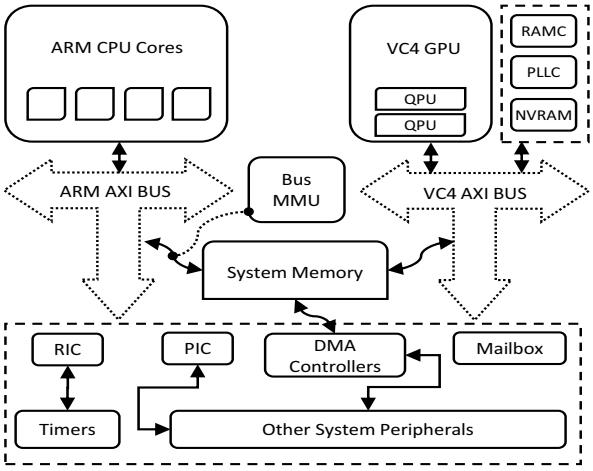

Figure 1. PI hardware platform architecture from a system security perspective. *Rounded rectangles denote system peripherals; dashed rectangles denote logical peripheral clusters.*

the physical system peripheral (§4.5). TIF allows us to efficiently implement memory, DMA and interrupt protections without the requirement of hardware support such as IO Memory Management Unit (IOMMU) and Generic Interrupt Controller (GIC) which are absent on the PI. UPI TIF also keeps the μHV complexity low since we don't resort to complex peripheral emulation and state maintenance.

UPI also leverages TIF in combination with initial trusted deployment to achieve a secure boot mechanism on the PI (§4.6;§4.7), an important feature that is currently not supported natively by the PI hardware.

The UPI architecture advocates the design and development of μHV extensions — called uberapps — that provide functional and security properties either in the context of the entire system or portions of the uberguest and applications running inside it (e.g., isolating a sensitive portion of the application). The concrete system properties and applications we developed are described in §3.2 and §6 respectively.

The UPI implementation, as of this writing, runs unmodified versions of the Raspbian Linux and Emlid real-time Linux [23] distributions on the PI with multi-core support (§5). UPI's μHV TCB is 5282 SLoC. The uberapps we developed are an additional 262 SLoC. The average runtime overhead for multi-core, computational and IO applications is 2%–6% (§7).

UPI is the first practical security architecture that has been implemented, deployed, and validated for the Raspberry PI. We are hopeful that UPI will spark the development of security sensitive applications on the PI, and inform its future hardware incarnations.

2. Slicing the PI

We begin by describing the platform architecture of the PI with an emphasis on system security. We then follow up with details on the startup sequence of the PI on power-up.

2.1. Hardware

The PI hardware has gone through three revisions since its inception. However, all revisions share a common System-on-Chip (SoC) logic from Broadcom with different ARM processors. The latest revision (v3) consists of a ARMv8 Cortex-A53 quad-core CPU with the Broadcom BCM2709 SoC and 1GB system memory. In this section, and the rest of the paper, we will focus primarily on the PI v3. Figure 1 shows the high-level platform hardware architecture of the PI from a system security perspective.

2.1.1. GPU Co-processor. The PI contains a 3D GPU based on the Video Core IV Architecture [15]. The GPU is a self-contained and highly automated co-processor and is internally divided into multiple instances of special-purpose floating-point processors termed a quad-processor (QPU). Each QPU consists of processor registers, ALU and instruction set spanning load, stores, branches and vector operations and capable of running full-fledged applications. The GPU interfaces with the ARM CPU cores (§2.1.2) via Mailboxes (§2.1.4) and accesses system memory via DMA (§2.1.7).

2.1.2. ARM Cores. The PI v3 contains a Cortex-A53 quad-core ARM processor which implements the ARMv8 architecture including hardware virtualization extensions [8], [9]. Each ARM core can operate in one of two overarching execution worlds: secure world and non-secure world. Within the non-secure world, there are three primary modes: EL2 (or hypervisor mode), EL1 (or guest kernel mode), and EL0 (or guest user mode). The EL2 mode supports translating guest physical addresses to system physical addresses via a second-stage page-table data structure.

2.1.3. System Bus and Memory View. The GPU and the ARM cores run on two separate buses. The buses use the standard ARM Advanced Microcontroller Bus Architecture (AMBA) [7] with the Advanced eXtensible Interface (AXI) fabric [6] to access peripherals via memory-mapped I/O. The GPU can access system memory and peripherals directly using a dedicated address map. The ARM cores on the other hand access system memory and peripherals via a coarse-grained system bus MMU which maps relevant ARM physical addresses into the GPU dedicated address map [13].

2.1.4. Mailboxes. The PI mailboxes are hardware conduits that allow communication between the ARM cores. This is used to synchronize the ARM cores during system bootup (§2.2) but can also be used for other operations that require synchronization (e.g., shared memory accesses). The PI has 16 mailboxes in total, 4 per each ARM core; each mailbox is a 32-bit wide register. Mailboxes allow communication between the ARM cores either with polling or via interrupts that can be configured in the interrupt controller (§2.1.6).

2.1.5. Timers. The PI platform hardware includes three timers: (a) the local timer is derived from the GPU clock

and supports four independent timers. Two of these are used by the GPU, one is reserved for the operating system and the other one is unused [13], [14]; (b) each ARM core has support for four architected 64-bit timers. EL3, EL2 and EL1 have one physical timer each, and there is one virtual timer attached to EL1 [9]; and (c) the watchdog timer includes a single counter which resets the hardware platform when enabled and counts down to zero[1].

2.1.6. Interrupt Controllers. The PI has two legacy interrupt controllers which are not virtualization aware [13], [14]. The peripheral interrupt controller (PIC) is responsible for handling interrupts from the GPU and other peripherals in addition to supporting certain special event interrupts (e.g., illegal bus access). The PIC is not vector based, but instead sets a bit for every interrupt that is pending. The PIC also lacks interrupt priority and it is upto the software to decide which interrupt to service. Finally, the PIC does not support automatic end-of-interrupt generation. Instead, the end-of-interrupt signal has to be sent by software directly to the device that triggered the interrupt.

The PI also has a root interrupt controller (RIC) that is responsible for handling architected timer interrupts, performance monitor interrupts, and mailbox interrupts for each ARM core. Further, the RIC includes support to route and trigger interrupts at a specified destination ARM core for timer and mailbox interrupts.

2.1.7. DMA Controllers. The PI has two direct memory access (DMA) controllers: (a) the legacy DMA controller and (b) the USB DMA controller[2]. Any system DMA has to be performed using either of the two DMA controllers[3]. The main DMA controller has 16 channels and allows both memory to memory and peripheral to memory transfers and vice versa. A non-USB peripheral is allocated a DMA channel and then performs the required DMA thereafter. The USB DMA controller is reserved for use for only USB transactions from host to USB device and vice-versa.

2.1.8. Other peripherals. In addition to the timers, mailboxes, Interrupt controllers and DMA controllers, ARM cores can access the following additional system peripherals: USB, PCM, I2C, SPI, GPIO, PWM, UART and Bluetooth. The GPU can access all the ARM accessible peripherals and in addition has dedicated access to the bus MMU, SDRAM control, PLLC and NVRAM.

2.2. System Startup

When the PI is powered on, the ARM cores are in reset state and the GPU boots up. At this point the system memory (SDRAM) is disabled. The GPU starts executing the first-stage bootloader from ROM in the SoC. The first-stage bootloader reads the boot-partition of the boot-media (e.g., SD card, USB) and loads the second-stage bootloader (`bootcode.bin`) into the GPU L2 cache, and transfers control to it. The boot-partition begins at a fixed address and is of fixed length regardless of the OS. `bootcode.bin` enables SDRAM, and loads and passes execution to the GPU firmware (`start.elf`). `start.elf` then initializes DMA, mailboxes and interrupt controller functionality required for GPU operation, sets up the bus MMU to allow ARM access to system memory, and boots up all the ARM cores in EL2 mode. All the ARM cores except for the boot-strap ARM core are then placed within a mailbox wait-loop. `start.elf` then loads the OS kernel image (`kernel.img`) transfers control to it on the boot-strap ARM core. Lastly, `kernel.img` gets control, boots up, loads all the required OS drivers and initializes the remaining ARM cores via a mailbox signal to establish a multi-core OS execution environment.

3. Goals, Properties and Assumptions

3.1. Goals

Our overarching goal is to enable design and development of performant security oriented applications on the PI to inject security properties in the existing platform and software stack. Our design goals fall broadly in three categories.

Commodity Compatibility and Unfettered Development: Our solution must integrate into the existing deployment ecosystem of the PI. It must be able to run unmodified stock operating systems and kernels and allow access to all programmable system peripherals (e.g., GPIO, I2C, SPI, USB, etc.). It must be generic enough to allow for a wide variety of security applications to be constructed.

Performance: Our solution must not preclude aggressive code optimization and must not adversely affect runtime performance. Further, commodity OS on multi-core hardware must be supported.

Low TCB and Low Complexity: Our solution should have a low TCB and complexity to facilitate manual audits and/or formal verification. Recent advances in formal verification has shown this is a critical requirement for verifiability [25], [46].

3.2. System Properties and Applications

The UPI architecture provides the following fundamental system security guarantees (Figure 2): (1) uberguest memory isolation and memory integrity of μHV and uberapps (§4.3); (2) μHV and uberapps liveness (§4.3.2); (3) μHV and uberapps memory protection from malicious sytem peripherals (§4.5) sans the hardware TCB (§3.4); and (4) secure boot (§4.7). We leverage these foundational system properties and showcase uberapps that we implemented which cover a

1. The watchdog timer is undocumented; the Linux driver bcm2835-watchdog source enabled us to figure out its design.

2. The USB DMA controller is undocumented; we had to scour through the Linux driver implementation to unearth the specifics.

3. The documents describing the DMA controllers do not mention how they integrate into the platform as a whole. We pieced together every peripheral datasheet and information publicly available to arrive at this conclusion.

System Properties	Architecture Mechanisms
1. Memory Integrity of μHV and uberapps	Memory Isolation (§4.3)
2. μHV and uberapps Liveness	Peripheral and Interrupt Partitioning (§4.3.2)
3. μHV and uberapps DMA protection	Trap-Intercept-Forward (§4.5)
4. Secure Boot (§4.7)	Trap-Intercept-Forward (§4.5)

Figure 2. UPI system properties and high-level architecture mechanisms. *All architecture mechanisms only rely on* PI*'s basic ARM h/w virtualization capabilities including h/w second-stage page-tables.*

wide spectrum of security applications spanning watchdog, attestation, secure storage and runtime monitoring (§6).

3.3. Non-goals

As the most prevalent (if not the de-facto) usage model of the PI is to run a single OS, we do not aim to run multiple operating systems or virtualize system resources in the traditional manner.

3.4. Attacker Model and Assumptions

We assume that the attacker does not have physical access to the PI. Our hardware TCB consists of the GPU, ARM, the interrupt controller and the DMA controllers. Other system peripherals and the OS kernel and applications are under the attacker's control. This is a reasonable assumption since a majority of attacks are mounted by malicious software or untrusted add-on boards interfacing via system peripherals. We assume that our hardware TCB is functionally correct. We also assume the correctness of the GPU first-stage and second-stage bootloaders and the GPU firmware to load `kernel.img` at boot-up. Section 8 discusses how some of these assumptions can be further relaxed.

4. Practical Security on the PI

We next describe our system architecture and how it addresses our goals and achieves our desired system properties (§3). We begin by briefly discussing alternate points in the design space and follow up with details of our security architecture.

4.1. Design Space

The competing approaches in the design space (c.f. related work §9) can be broadly divided into:

Hardware Containers: ARM TrustZone [10] allows software to run in an isolated compartment in the processor secure-world. However, TrustZone requires the use of memory exclusive to the secure-world, which is absent in the PI. Further, the TrustZone architecture does not include any support for runtime monitoring of the OS executing in the normal-world.

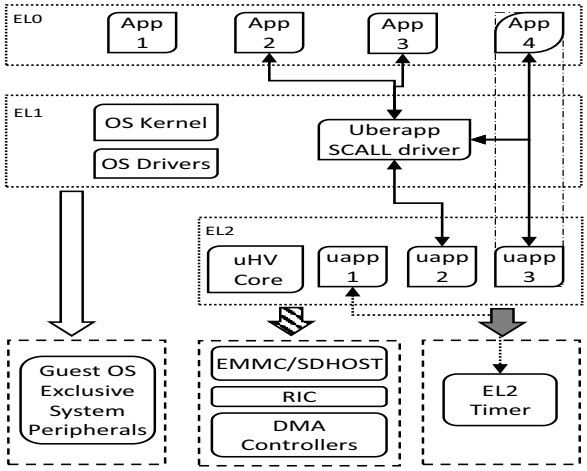

Figure 3. UBERPI micro-hypervisor based system security architecture. *Dashed boxes indicate logical peripheral groups; Dotted boxes indicate operating privilege levels (EL0, EL1, EL2); Dashed dotted box indicates a regular application split and bound to an exclusive uberapp; Thick solid down-arrow indicates exclusive peripheral access by the* UPI *framework; Hollow down-arrow indicates exclusive peripheral access by the uberguest; Patterned down-arrow indicates peripheral accesses that are subject to trap-inspect-forward; Solid bi-directional thin arrows indicate synchronous uberapp calls; Dotted bi-directional thin arrow indicate asynchronous uberapp calls.*

OS Containers: OS namespace and kernel support can be leveraged to implement sandboxing mechanisms [3], [18] and type-2 hosted hypervisors [19]. However, these approaches end up including a large portion of the OS into the TCB. Further, the isolated containers are resource constrained without full access to platform hardware.

Baremetal VMMs: ARM general-purpose baremetal hypervisors [34], [50] support multiple guests. However, they rely on hardware support for IO Memory Management Unit (IOMMU) and virtualizable Generic Interrupt Controller (GIC) which are absent on the PI. This in turn prevents them from achieving adequate protection (e.g., DMA) or allowing full access to system peripherals within the guest. Further, they lack support for commodity OSes and only support Linux kernels on an stripped down emulated hardware platform.

4.2. UBERPI (UPI) Architecture Overview

To achieve our design goals and system properties (§3; Figure 2), we propose a micro-hypervisor (μHV) based system security architecture called UBERPI. In stark contrast to traditional hypervisor architectures [41], [45], [46] that rely on several foundational hardware primitives (IOMMU, GIC, and hardware root-of-trust) typically found on high-end hardware platforms (e.g., ARM and x86 server platforms), UPI achieves our goals by leveraging just basic platform features found on the low-cost PI. Figure 2 summarizes the UPI system properties and architecture mechanisms.

The high-level UPI system architecture (Figure 3) is based on three core concepts: (a) μHV core and uberguest: most system peripherals are controlled directly by a single,

commodity, unmodified OS (uberguest) achieving high performance while still ensuring strong isolation and runtime protection; (b) uberapps: act in the context of the uberguest or uberguest applications to provide required security properties; (c) μHV Trap-Inspect-Forward: facilitates secure boot and runtime protection of the μHV core and uberapps via light-weight peripheral and memory access interceptions.

4.3. μHV Core and Uberguest

The μHV core forms the heart of the UPI architecture and includes supporting libraries that sit directly on top of the platform hardware. The design choice of the uberguest being a single full-featured commodity unmodified OS fits squarely with the PI's de-facto usage model and development ecosystem. This choice also allows us to greatly minimize μHV core complexity and consequently TCB since most system peripherals are directly handled by the uberguest. Further, the uberguest model results in high performance since all peripheral interrupts are directly handled and serviced by the OS without any μHV core intervention. Section 5.2 describes the uberguest implementation in more detail, along with challenges of handling guest memory reporting and multi-core enablement.

4.3.1. Uberguest Memory Isolation. A malicious uberguest can directly access UPI memory regions thereby compromising system security. UPI leverages the PI Cortex-A53 hardware virtualization second-stage page tables for uberguest memory isolation. The second-stage page-tables are resident within the μHV core memory regions and ensure that uberguest physical memory accesses are translated to the actual system physical address via a hardware second-stage page-walk. The UPI memory regions are marked inaccessible within the second-stage page-tables which will cause the hardware to disallow any direct memory access to the UPI memory regions by the uberguest.

4.3.2. μHV Core Peripheral and Interrupt Partitioning. Certain security applications may entail reserving specific system peripherals for exclusive use by the μHV core and/or the uberapps and the subsequent handling and servicing of their interrupts, e.g., a dedicated system timer for secure periodic processing.

UPI handles such peripheral mappings using the hardware second-stage page tables described previously, to ensure that the uberguest does not see or have unrestricted access to the peripheral that is reserved for (exclusive) use by UPI. However, the lack of GIC hardware on the PI precludes interrupt virtualization and multiplexing making interrupt partitioning challenging.

UPI leverages two key insights to allow interrupt partitioning without the GIC and complex peripheral emulation. First, the ARM architecture supports two overarching interrupt mechanisms (fast and regular), but commodity OSes only make use of regular interrupt mechanism. Second, the PI RIC, which supports routing of system peripheral interrupts via either of the aforementioned mechanisms is not used for system peripherals; only the PIC is used. UPI therefore uses a combination of fast interrupts and the RIC fast interrupt routing to achieve efficient interrupt partitioning. Implementation details in the context of an exclusive timer peripheral can be found in §5.2.2 and §5.3.2.

4.4. Uberapps and Uberapp Interactions

The μHV core interacts with the uberguest via the well-defined ARM hardware virtualization platform interface [8]. In UPI these interactions are serviced by the μHV core (e.g., guest memory reporting) or handled by a μHV core extension that we term uberapps. For example, tracking uberguest process contexts for application privacy or a watchdog application for ensuring keep-alive sensitive operations in the face uberguest failures or breach.

UPI also advocates a development ecosystem where a regular uberguest application can have its sensitive portions isolated as uberapps which are isolated from the remainder of the uberguest and other applications. For example, storing sensitive signing keys for an encrypted file system or for platform attestation. Uberapps can initiate and maintain trust with components higher in the stack (e.g., untrusted uberguest application) via existing and complementary application specific mechanisms. e.g., uberguest application code/data isolation [47] and/or (property-based) attestation [11].

Uberapps can be synchronous (e.g., invoked via a synchronous call; SCALL) or asynchronous (e.g., executed periodically). UPI uses a combination of hardware virtualization traps and ARM architected physical timers to support both uberapp execution models (§5.3). This allows for a wide range of security applications to be developed in practice (§6)

Note that for synchronous uberapp communication, the uberapp carries the onus of data sanitization for input parameters (e.g., range checks) since it is processing data from a potentially compromised guest. The uberguest memory isolation setup by UPI (§4.3.1, §5.2.1) ensures that a compromised uberguest cannot influence the uberapp data sanitization process.

4.5. Protections via Trap-Inspect-Forward

UPI uses hardware second-stage page-table protections and hardware virtualization traps for light-weight trap-inspect-forward (TIF) where peripheral accesses are selectively *trapped* and *inspected* for correct accesses before *forwarding* the access directly to the physical system peripheral. UPI TIF allows us to implement various system protections as described below without requiring hardware IOMMU and GIC support (absent in the PI) and without resorting to complex peripheral emulation and state maintenance. Our evaluation shows that TIF results in low-TCB and incurs minimal performance overhead (§7).

DMA Protection: A malicious guest or system peripheral can mount DMA style attacks [38] in order to compromise UPI memory integrity. As described previously in §2.1.7

any DMA request on the PI has to be performed using the platform DMA controllers. UPI's DMA protection mechanism monitors the DMA controllers to prevent unauthorized DMA requests to UPI memory regions thereby ensuring that malicious peripherals cannot compromise UPI. More specifically, UPI leverages TIF on the PI DMA controller register space to prevent any form of DMA attacks in the system; §5.4.1 describes further implementation details.

Interrupt Protection: A buggy or malicious guest can tamper with the RIC to disable fast interrupt routing to UPI resulting in denial of service type attacks. §5.4.2 describes how UPI leverages TIF to protect the RIC register space to preserve fast interrupt routings setup by the UPI μHV core and/or uberapps.

4.6. UBERPI Lifecycle

Installation: End-users receive the PI pre-loaded with the boot-partition (§2.2) image from the UPI distributor (described later in this section). System developers on the other hand can receive the UPI installation kit, consisting of a signed boot-partition image from the UPI distributor, which they verify using the public key and copy it to the boot-partition of the PI. Then the user optionally (re-)configures the UPI μHV core and uberapps, e.g., compile a custom binary image with required uberapps from a signed source blob. UPI is now ready for startup and normal operation

Startup and Recovery: On bootup UPI μHV core is loaded which in turn initializes the uberapps and eventually boots the uberguest. Normal system operation is characterized by uberguest execution interspersed with μHV core and uberapp interactions. When the uberguest is shutdown, the UPI μHV core gets control and cleans up required internal state including those of uberapps before powering down the PI. Malicious uberguest behavior (e.g., writing to UPI memory regions, performing DMA to UPI memory regions, etc.) are disallowed gracefully by ignoring such actions. However, such actions can also be handled via a dedicated uberapp for required user signaling (§6).

UPI Distributor: The role of the UPI distributor might be played by a trusted company or organization, such as IBM, Microsoft, Google. UPI's key insight is that by agreeing to install and use UPI, the user is expressing their trust in the UPI distributor, since UPI will be operating with maximum privileges. To some extent, OS distributions today already implicitly operate on this assumption. When a user installs a distribution they also trust those distributions to provide software that does not contain malware.

Updates: As UPI components (μHV core and uberapps) evolve, they need to be updated. This process is straightforward since the UPI distributor can simply release a signed list of the update and/or source binary to be installed as described in the beginning of this section.

4.7. Secure Boot

The UPI lifecycle ensures a valid PI boot-partition image to begin with (§4.6). However, malware in the uberguest can tamper with the binary images on the boot-partition to subvert subsequent loading of the framework. Secure boot is one such mechanism to prevent such attacks on embedded platforms and ensure load-time integrity; the SoC boot-ROM begins a signed execution chain of boot-loaders until the final kernel is loaded. However, the PI boot-ROM does not currently implement such a facility, although the SoC hardware itself has all the necessary capabilities (§8).

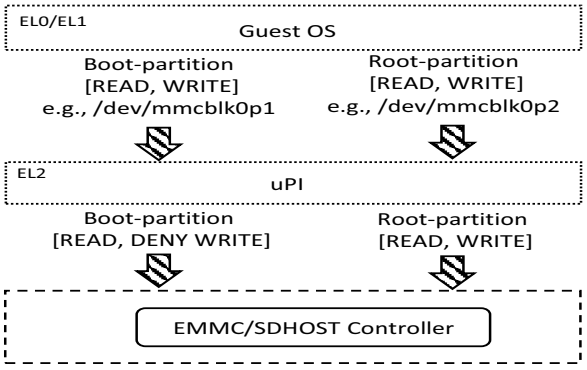

Figure 4. UBERPI secure boot leverages intercept and inspect mechanism (indicated by Patterned down-arrow) on the MMC and SDHOST controller in order to deny writes to the boot partition containing UPI boot binaries while passing through all other accesses.

UPI uses a combination of trusted deployment (§4.6) and trap-inspect-forward (§4.5) to achieve secure boot on the PI. The UPI lifecycle ensures that a valid boot-partition image is in place (§4.6). This ensures that the integrity of the chain of execution from the bootrom SoC to the UPI framework binary is valid to begin with. UPI relies on a key insight towards preserving the integrity of the boot-partition contents. The boot-partition on the PI is of a fixed length and only used by the GPU to load the boot-loaders and the final `kernel.img` and is not used by the OS thereafter.

Figure 4 shows how UPI achieves secure boot on the PI. UPI takes advantage of the fact that the PI storage is managed by a EMMC/SDHOST controller which implements the SDIO protocol and directs storage operations to the attached storage device (e.g., SD card). When the uberguest writes to the SDHOST controller registers for read and write operations, UPI intercepts only the write operations and inspects the SDHOST registers to ensure that the target sector addresses do not belong to the boot-partition. If it does, it denies the write. Our evaluation shows that such interpositioning has minimal effect on the performance (§7).

4.8. UBERPI Security Analysis

We now present a security analysis of the UPI architecture and properties (§3.2; Figure 2) in the context of our attacker model and system assumptions (§3.4).

UPI's use of hardware second-stage page tables ensures that code executing within the uberguest cannot directly address the μHV core or uberapps, thus protecting their secrecy and integrity. Thus, even though a uberguest can be exploited via software or network vulnerabilities, such

exploits cannot compromise the μHV core or uberapps directly.

UPI's support for periodic uberapps (§4.4) and interrupt protection (§4.5) prevents malware in the uberguest from carrying out a denial of service (DoS) attack on UPI. Note that malware in the uberguest can mount a DoS attack on synchronous uberapps by compromising the uberguest application that is bound to the uberapp. However, such DoS attacks can be detected by employing a watchdog uberapp (§6).

UPI's DMA protection mechanism (§4.5) ensures that malicious peripherals (e.g., USB, SPI or I2C peripherals) can only access uberguest memory regions, and prevents advanced DMA controller based attacks such as DMA gadgets [38].

Finally, UPI's secure boot mechanism (§4.7) ensures that the uberguest cannot write to the boot-partition of the PI. With the UPI installation ensuring a legitimate boot-partition to begin with (§4.6), this ensures a valid UPI image will always be loaded during boot-up.

5. UBERPI Implementation

The UPI prototype implementation described in this section runs the unmodified Raspbian 32-bit Linux distribution shipped with the PI v3 with version 4.4.y of the Linux kernel. In addition, UPI can also run the unmodified Emlid real-time Linux kernel [23]. The implementation builds on top of the open-source UBER-eXtensible Micro-Hypervisor Framework [45], [46] and has a 24MB runtime memory footprint.

5.1. Bootup

The PI boot-partition `kernel.img` (§2.2) is replaced by a unified UPI binary image consisting of: the UPI trampoline code; the original unmodified (uberguest) `kernel.img`; and the UPI μHV core and uberapp bundle. The UPI trampoline essentially transfers control to the UPI μHV core which in turn prepares the platform for uberguest and uberapp execution.

5.2. μHV Core and Uberguest

The UPI μHV core performs the following operations: (a) initializes μHV EL2 page-tables and the ARMv8 platform hardware virtualization support; (b) sets up memory, DMA, interrupt and boot protections; and (c) transfers control to the uberguest kernel to start the OS boot process.

5.2.1. Uberguest Memory Isolation. UPI uses ARMv8 support for second-stage page-tables to implement uberguest memory isolation. The second-stage page-tables is a 3-level structure describing the guest physical address mappings to actual system addresses with additional protection bits (device, no-access, read-write etc.). UPI framework regions (where the second-stage page-tables themselves reside) are marked no-access while other guest memory regions are marked read-write-execute. The VTCR register is then set to activate a 3-level page-table format with appropriate shareability and cacheability attributes (inner shareable and write-back, write-allocate caching). Finally, the VTTBR register is loaded with the base of the second-stage page-tables and second-stage page-table translation is enabled via the HCR register.

5.2.2. μHV Core Peripheral and Interrupt Partitioning. In the ARM ecosystem all system peripheral accesses happen via memory-mapped IO (MMIO). UPI, by default maps all system peripherals with read-write protections except for the RIC, DMA controllers and the MMC/SDHOST controller which are setup as described in §5.3.2, §5.4.1 and §5.5 respectively. UPI uses the hardware second-stage page-tables for such mappings. Note, other system peripherals can be setup on demand if required to be used exclusively by the UPI framework and uberapps (cf. `wdog` uberapp and the hardware watchdog peripheral; §6).

ARM uses FIQ signaling for fast interrupts and IRQ signaling for normal interrupts. The μHV core programs the HCR register to indicate no-trapping on IRQs. This allows uberguest to handle all peripheral interrupts without any intervention by UPI. UPI also sets the FIQ redirection bit in the HCR register for fast interrupt redirection to the μHV core. When this bit is set, hardware transfers control to a set location in EL2 mode on FIQ interrupts. The corresponding peripheral interrupt is then handled and cleared within the μHV core and/or the uberapp as required.

5.2.3. Uberguest Memory Reporting. A native ARM OS during its boot-up on the PI has the option of using either the Device Tree Blob (DTB) or ATAGS in order to obtain the system memory map[4]. The DTB and ATAGS are essentially flat data structures that contain information about system memory and devices and various memory regions and their attributes (e.g., device MMIO, usable memory and reserved). The DTB and ATAGS are setup by the GPU firmware prior to loading `kernel.img`.

However, with UPI loaded there must be a way to report a reduced memory map excluding the UPI memory regions to the OS. If not, the OS at some point during execution will end up accessing the UPI framework which would cause a fault since UPI memory is marked no-access within the second-stage page-tables (§5.2.1). UPI revises the DTB and the ATAGS structures adding entries to mark the UPI memory regions reserved. Thus a well-behaved OS will not attempt to access the UPI memory regions during the lifetime of its execution[5].

4. Linux kernels adhere to this requirement. However, some commodity OSes may use boot-loaders which may not adhere to this requirement (e.g., Windows IoT core). The work-around in such a case is to modify the boot-loader to use the ATAGS/DTB instead.

5. A malicious OS can still try to access the UPI memory regions, but will cause a fault in the second-stage page-tables; currently this causes UPI to ignore the access and resume the OS.

5.2.4. Multi-core Support. On the PI only one ARM core called the boot-strap core is started when the GPU firmware transfers control to `kernel.img`. The other (application) cores spin on the core mailbox waiting for a signal to awaken. At some point during the boot process of a native OS, the kernel will signal the mailbox which causes the other cores to awaken and start executing kernel code. UPI on boot-up initializes all the cores to EL2 mode and leaves all the application cores spinning on their mailbox and letting the boot-strap core start the OS in EL1 mode. When the OS signals the mailbox, the cores spinning in EL2 mode respond by grabbing the starting address (written to the mailbox) and transfers control to the OS in EL1 mode at the starting address.

5.3. Uberapps and Uberapp Interactions

5.3.1. Synchronous Uberapp Interactions. ARMv8 hardware virtualization traps provide a hardware enforced mechanism for synchronous uberapp interactions. The HVC instruction is used to perform a hypercall and is used as a primary means for synchronous uberapp interactions from the uberguest. Other synchronous uberapp interactions happen via hardware assisted trap mechanisms including second-stage page-faults (as a result of protection violation in the second-stage page-tables) and designated instruction traps (e.g., execution of system instructions). Upon all such traps, the μHV core gets control, marshals required parameters and transfers control to the corresponding uberapp handlers.

5.3.2. Asynchronous Uberapp Interactions. The PI Cortex-A53 ARM processor has support for per-core physical timers. The physical timers are banked across all the operating modes. The EL2 mode physical timer is controlled via a group of system registers which include the timer-value register (CNTHP_TVAL), compare value register (CNTHP_CVAL) and a control register (CNTHP_CTL). The timer-value is the physical timer and is incremented every clock cycle. The control register is programmed to trigger an interrupt if it matches the compare value register, UPI programs the RIC to generate a FIQ interrupt for interrupts received via the EL2 mode physical timer. The FIQ interrupt handler within the μHV core is responsible for invoking the corresponding uberapp timer handler for any periodic processing.

5.3.3. Uberapp API. The UPI μHV core implementation currently provides the following application programming interface (API) to uberapps: (a) manipulating second-stage page-table protections – allowing uberapps to set appropriate memory protection on uberguest memory pages; (b) enabling platform h/w virtualization features – allowing uberapps to activate appropriate uberguest event reporting mechanism (e.g., trap control register accesses); and (c) installing platform trap handlers – allowing uberapps to install their custom trap handlers (e.g., for timer processing). The aforementioned set of APIs allow us to implement several practical security applications as described in §6.

5.4. Protections via Trap-Inspect-Forward

5.4.1. DMA Protection. As described previously (§2.1.7) the PI contains a legacy DMA controller and a USB DMA controller on the SoC.

Legacy DMA Controller: The legacy DMA controller contains 16 DMA channels and each channel is interfaced via a pair of registers: the control block address register and the status and enable register. The DMA control block (`dmacb`) structure consists of the source and destination DMA physical addresses along with the length of transfer and the address of the next control block structure. This way multiple control blocks can be linked in order to perform batch DMA operations. The legacy DMA controller is MMIO mapped in the second-stage page-tables with a no-access protection. This allows trapping on both reads and writes to channel pair registers.

UPI uses DMA control block shadowing in order to protect framework memory from (malicious) DMA transactions. On write to the `dmacb` address register, we iterate through the control block list supplied by the OS and copy it over to a μHV core control block area. During this copy we also ensure that the control block source and destination do not include any UPI memory regions. We then set the `dmacb` address register to the address of the shadow control block. Subsequent write to the DMA enable register by the OS will then use the shadow control block for DMA transfers. Similarly, reads to the control block address will return the original control block address as expected by the OS. This shadowing mechanism is both efficient (§7) and effective in terms of preventing any form of DMA attacks including DMA gadgets [38].

USB DMA Controller: The PI also consists of a USB DMA controller (`usbdmac`) which is part of the USB OTG host controller. The `usbdmac` consists of various registers which form a DMA descriptor. One such address is the host-address which is used to transfer USB data into or out of the system. The `usbdmac` is mapped as MMIO in the second-stage page-table with read-only protection. This allows us to pass through descriptor reads any trap only on writes. On write trapping via second-stage page-faults we check to ensure that the value written to the host address field is part of the uberguest memory region. If not, the write is denied.

5.4.2. Interrupt Protection. The PI root interrupt controller (RIC) is employed for interrupt partitioning for peripherals that are reserved exclusively for the μHV core and/or uberapps. The EL2 timer is one such example which is used by the μHV core for periodic uberapp processing (§5.3). The RIC contains an interrupt enable and interrupt type field for physical timers as well as other system peripherals.

UPI maps the RIC as MMIO and with read-only protections in the second-stage page tables. This allows us to intercept on writes to the RIC registers while allowing reads to pass through. The current implementation ignores any writes to the interrupt enable and interrupt type fields for the EL2 timer peripheral. This ensures that EL2 timer interrupts for periodic uberapp processing are always fired.

5.5. Secure Boot

The UPI secure boot implementation supports any SD card that is complaint to the SDIO v3 standard. This is the same requirement imposed by the PI itself. As per the SDIO specification, before any operation (read or write) is performed on the card, the MMC/SDHOST controller ARG register is set to the actual sector address for performing the operation and the BLKCOUNT register is set to the number of blocks that needs to be taken into account. This is then followed by a write to the SDCMD register which specifies if a read (17/18) or write (24/25) operation is to be performed. For example, to write a sector at address 0 on the SD card the ARG register is set to 0, BLKCOUNT register is set to 1 and the SDCMD register is set to 24.

UPI maps the SD HOST controller register space as MMIO and read-only in the second-stage page-tables. This allows us to only trap on writes to the register space allowing reads to pass through thereby preventing unnecessary traps due to any status condition reads via the register address space. Upon a write to the SDCMD register for a write command, UPI checks the ARG and the BLKCOUNT register to ensure that they do not fall within the boot-partition sector range. As described in §2.2, this is a fixed range for the PI regardless of the running OS. If UPI detects writes that fall within the boot-partition sector range, it ignores the write and sets the status register to indicate an error, else the write command is allowed to go through.

6. UBERPI Applications

In this section we describe uberapps we have implemented using UPI. The uberapps serve to illustrate the range of security applications that can be realized using the UPI framework.

Watchdog: Any security posture benefits form the use of a non-circumventable watchdog application. We developed a uberapp called `wdog` which reacts to a malicious uberguest event (e.g., access to UPI framework memory) or a frozen uberguest by restarting the PI. `wdog` employs trap-inspect-forward (§4.5) on the hardware watchdog peripheral by marking the peripheral register space read-only in the second-stage page-tables. `wdog` services hardware second-stage page-faults traps that occur on uberguest writes to the watchdog peripheral and resets a zero-initialized internal counter. `wdog` also services second-stage page-faults in response to malicious uberguest memory accesses and maintains a status variable indicating such accesses. Finally, `wdog` relies on periodic uberapp execution to continuously increment its internal counter. If the internal counter reaches a predefined value or if the status variable indicates any malicious accesses on each periodic servicing, `wdog` resets the system.

Micro Trusted Platform Module (TPM): Another important security application is attestation. We implemented a micro TPM uberapp that is based on the TrustVisor open-source x86 micro-TPM library implementation[6]. Our implementation separates the user-mode micro-TPM test application and library operations of PCR extend, read, seal and unseal along with related private keys into the `utpm` uberapp. The `utpm` uberapp interfaces with the rest of the micro-TPM test application via synchronous uberapp calls. The `utpm` implementation currently uses ephemeral keys, but long-term storage can be achieved by using the protected boot-partition and/or the PI NVRAM (§8).

Encrypted File System: Secure storage provides secrecy, integrity and/or freshness for a software module's data at rest. We implemented an encrypted file system based on the open-source FUSE-based pa5-encfs[7], to support encrypted storage within the uberguest. We isolated the sensitive portion of the original pa5-encfs implementation which corresponds to the private key and password operations into the `encfs` uberapp. `encfs` communicates with the remainder of the pa5-encfs implementation via synchronous uberapp calls.

Contextual Inspection: Lastly, to demonstrate a class of runtime monitoring application, we developed a stand-alone uberapp called `ctxtrace` to perform uberguest wide process tracking. `ctxtrace` relies on synchronous uberapp invocation via hardware virtualization traps on instruction execution. More specifically `ctxtrace` traps on TTBR0 and TTBR1 system register writes to track process page-table (context) switches. Upon such writes, `ctxtrace` logs the process id and its corresponding page-table base for future inspection. `ctxtrace` can be used as a foundation for developing more full-fledged process privacy preserving applications or simply for tracing and debugging purposes.

7. UBERPI Evaluation

In this section we evaluate our UPI implementation using three metrics: code size, development effort and performance. For brevity we focus on the 32-bit Raspbian OS (Linux kernel 4.4), the default OS distribution bundled with the PI.

7.1. Trusted Computing Base (TCB)

Like all security systems, UPI must assume the correctness and security of its components. One way to make this assumption more likely to hold is to keep things modular and reduce the amount of code and complexity that must be trusted. This in turn reduces the opportunity for bugs. We use the `sloccount` utility to measure the code size and composition of our prototype. UPI's TCB comprises of runtime libraries, μHV core and uberapps. The runtime libraries currently include a tiny C, crypto and micro-TPM library. The μHV core comprises of base platform code, uberguest and uberapps support, secure boot, memory, DMA and interrupt protection mechanisms. Combining everything, UPI's SLoC is 5544. UPI's SLoC and modular implementation is well within range for state-of-the-art system software verification approaches [25], [45], [46] to be readily applied

6. http://xmhf.org

7. http://github.com/ianks/fuse-encrypted-filesystem

Component	.c	.s (SLoC)	.h
Runtime Libraries:			
C library	153	0	341
Crypto library	1819	0	180
Micro-TPM library	345	0	101
uberapp `SCALL` library	90	0	15
μHV core:			
Platform support	99	620	435
Uberguest support	465	0	69
Uberapp support	103	0	10
Memory protection	145	0	0
DMA protection	153	0	29
Interrupt protection	37	0	0
Secure-boot	73	0	0
uberapps:			
Context Tracer (`ctxtrace`)	74	0	5
Watchdog (`wdog`)	23	0	0
Encrypt FS (`encfs`)	50	0	23
Micro-TPM (`utpm`)	57	0	30
	3686	620	1238
Total		**5544**	

Figure 5. UPI Trusted Computing Base: Contains modular components comprising runtime libraries, μHV core and uberapps. *.c = C source; .s = Assembly source; .h = header file*

to the code base for higher assurance. We leave this task for future work. UPI's SLoC is also an order of magnitude smaller than other ARM hypervisors advertised to work with the PI (Xvisor [34]: approx. 265K SLoC; KVM/ARM [19] within Linux Kernel: in millions of SLoC).

7.2. Development Effort

Development of the UPI prototype including uberapps took 6 person months in total. Majority of this time (5 person months) was spent towards the implementation of runtime libraries and the μHV core, which is a one time effort that will continue to proceed in an incremental fashion as the framework evolves. An additional person month was spent on developing all the uberapps. UPI's modularity greatly facilitated rapid development especially those of the uberapps which relied on the μHV core and runtime libraries for their functionality. Re-factoring regular applications to include a uberapp counterpart (e.g., `utpm` and `encfs`) consists of changing the application structure to isolate and bridge the sensitive parts with a corresponding uberapp. The latter is trivial via uberapp interfaces provided by UPI while the former incurs modest development costs; it took us 3 person weeks to re-factor the original micro-TPM and encryptFS C source base to run it within UPI with sensitive portions isolated as uberapps.

7.3. UPI Micro-benchmarks

We designed and ran a number of micro-benchmarks to quantify important low-level interactions between the UPI framework (including μHV core and uberapps), the platform

HVC	HVC-SCALL	NestedPF	InstTrap	IntTrap
252	3996	345	360	297

Figure 6. UPI micro-benchmarks quantifying low-level interaction primitives between UPI (including uberapps), the platform hardware and the uberguest. All values are in clock-cycles.

hardware, and the uberguest. A primary performance cost is the time spent on transitioning between the uberguest and the UPI framework. This includes transitions due to hypercalls, second-stage page-faults, interrupt and instruction traps. We designed a custom Linux kernel driver which ran in the uberguest and executed the micro-benchmarks. Measurements were obtained using cycle counters on a single core configuration to ensure consistency and reduce measurement variability in the context of multi-core. Instruction barriers were used before and after taking timestamps to avoid out-of-order execution or pipe-lining from skewing our measurements. Figure 6 lists the transition costs between the uberguest and the UPI framework for various classes of traps. The costs are minimal as ARM provides banked register and state support for the EL2 mode. They also compare favorably to high-end ARM server platforms [19]. Figure 6 also lists the transition cost of a synchronous call as performed by a uberapp (HVC-SCALL). This includes cost due to pinning and unpinning uberguest memory buffers, invoking the kernel driver, and saving and restoring registers within the μHV core. This overhead is also comparable to existing Type-1 and Type-2 ARM hypervisors running on high-end server platforms [19].

7.4. Uberguest Benchmarks

When the uberguest is operating without any uberapp interactions, there is memory, interrupt, DMA and secure-boot protection overheads at a low-level. We measure these overheads using micro-benchmarks and application benchmarks on the uberguest.

7.4.1. Uberguest Micro-benchmarks. We use the `lmbench3` micro-benchmarks to measure the memory and interrupt protection overheads in general. The benchmarks measure uberguest operations such as system calls, interrupt handling, local communications (pipes, sockets, etc.), context switches and memory management primitives. Figure 7 compares the UPI uberguest `lmbench3` micro-benchmark overheads with the native system without UPI. The overheads in most cases are small with the exception of fork, exec and mmap which stress the hardware second-stage translation tables and caches and consequently incur higher overheads. However, the overheads are still within reasonable bounds and compare favorably with other ARM hypervisors running on high-end hardware [19].

To measure UPI DMA protection and secure-boot protection overheads, we use the industry standard `iozone` and `netperf` micro-benchmarks to measure overhead of MMC/SD card and the network operations respectively. We use `iozone` in auto mode for reads and writes to a 16MB

Benchmark	Native	UPI	Overhead
Processes – times in microseconds			
null call	0.51	0.51	0.00
null io	0.54	0.55	0.01
slct TCP	21.60	21.70	0.10
sig inst	0.79	0.81	0.02
sig hndl	3.51	3.55	0.04
fork proc	528.00	573.00	45.00
exec proc	4920.00	5147.00	227.00
sh proc	9975.00	10000.00	25.00
Context Switching – times in microseconds			
2p/0k ctxsw	6.00	7.00	1.00
2p/16k ctxsw	5.90	6.20	0.30
2p/64k ctxsw	5.00	6.00	1.00
8p/16k ctxsw	6.50	7.20	0.70
8p/64k ctxsw	22.90	23.10	0.20
16p/16k ctxsw	7.40	8.60	1.20
16p/64k ctxsw	24.90	27.90	3.00
MM System latencies – times in microseconds			
mmap	13800.00	19000.00	5200.00
prot fault	0.49	0.54	0.05
page fault	1.42	2.09	0.67
100fdselect	7.60	7.60	0.00
Local comms. latencies – times in microseconds			
pipe	19.50	21.70	2.20
AF UNIX	17.50	17.50	0.00
UDP	37.70	39.90	2.20
TCP	48.00	50.70	2.70
TCP conn	78.00	80.00	2.00
Local comms. bandwidth – in MB/second			
pipe	632.00	614.00	18.00
AF UNIX	1915.00	1914.00	1.00
TCP	454.00	430.00	24.00
mmap reread	1688.00	1642.00	46.00
bcopy (libc)	1052.00	1044.00	8.00
bcopy (hand)	1052.00	1050.00	2.00
mem read	1698.00	1672.00	26.00
mem write	1272.00	1268.00	4.00

Figure 7. UPI uberguest `lmbench3` low-level uberguest OS benchmarks and comparison with native system without UPI

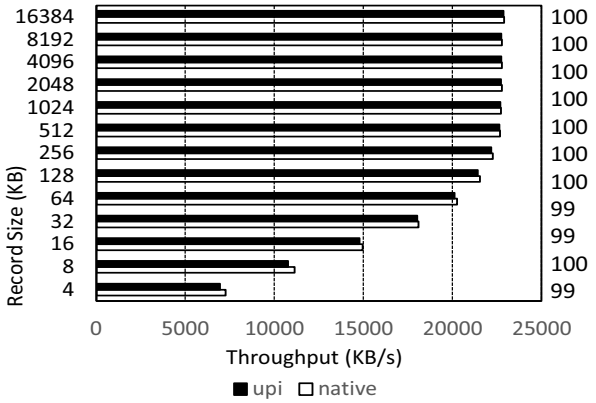

Figure 8. UPI uberguest MMC/SD card `iozone` disk read microbenchmarks. *Secondary axis: % of native system performance.*

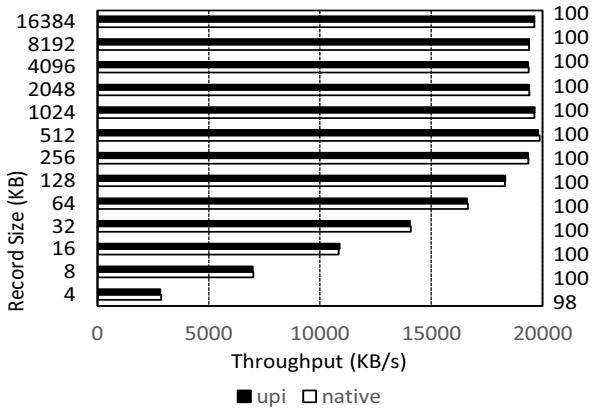

Figure 9. UPI uberguest MMC/SD card `iozone` disk write microbenchmarks. *Secondary axis: % of native system performance.*

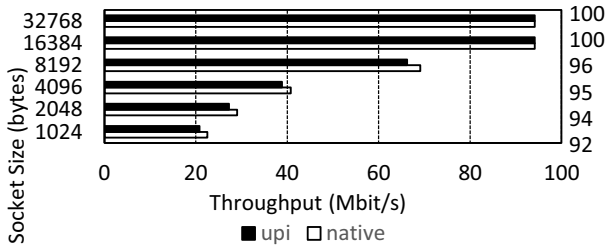

Figure 10. UPI uberguest `netperf` TCP_STREAM network microbenchmarks. *Secondary axis: % of native system performance.*

test file with the direct IO option to reduce measurement variability due to file-system caches. The TCP_STREAM and UDP_STREAM `netperf` benchmarks are used to measure the low-level network performance for various socket and message sizes starting from the OS supported minimum sizes.

The `iozone` micro-benchmarks (Figure 8 and Figure 9) mostly run close to native speeds at higher record sizes and have minimal overheads (0-2%) at lower record sizes. A similar trend is observed with the `netperf` micro-benchmarks (Figure 10 and Figure 11) which run close to native speeds at higher socket and message sizes and incur small overheads at lower socket and message sizes. The overheads at lower record, socket, and message sizes are due to multiple DMA transfers which incur control block shadowing overheads. We note that the SDHOST trap-inspect-forward incurs minimal overhead since most transactions employing DMA use sparse writes to the SDCMD register.

7.4.2. Uberguest Application Benchmarks. We execute computational and memory benchmarks from the Phoronix Test Suite for Linux [35]. We use pts/ramspeed, pts/-cachebench and the pts/scimark suites as they provide a good mix of computational and memory intensive application benchmarks. Figure 12 shows the execution of these benchmarks on the UPI uberguest as % of the native system's performance without UPI. The average overhead of the

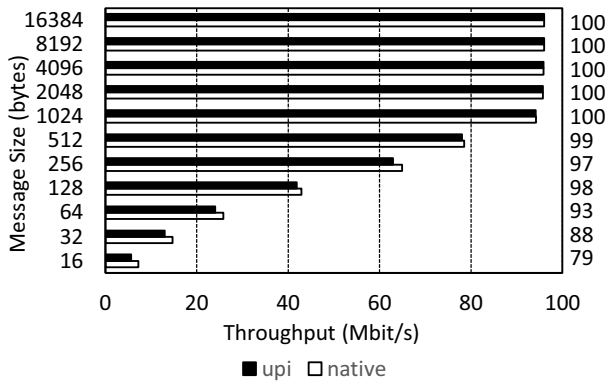

Figure 11. UPI uberguest `netperf` UDP_STREAM network microbenchmarks. *Secondary axis: % of native system performance.*

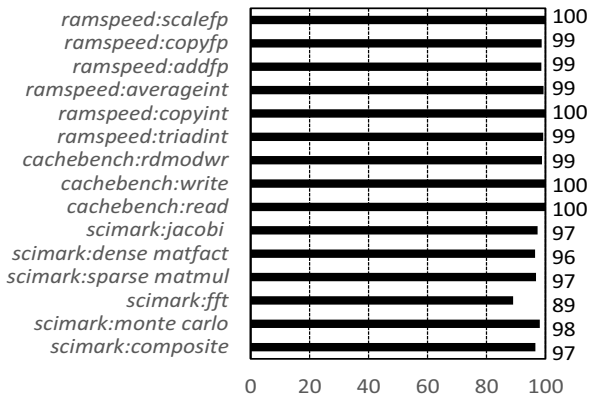

Figure 12. UPI uberguest computational and memory benchmark execution as % of native system performance without UPI.

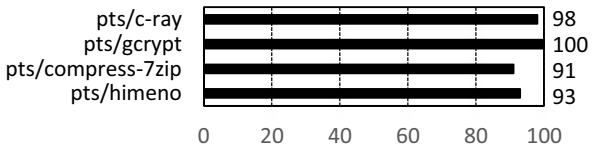

Figure 13. UPI uberguest multi-core benchmark execution as % of native system performance without UPI.

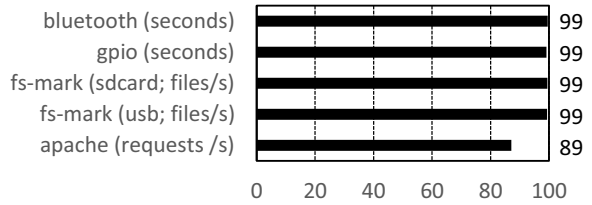

Figure 14. UPI uberguest IO benchmark execution as % of native system performance without UPI.

benchmarks is only 2%. scimark:fft is the only benchmark incurring a relatively larger overhead of 11%. We attribute this to the overhead to hardware second-stage page-tables and caches.

To test the uberguest multi-core performance, we use the multi-core Phoronix benchmarks: pts/c-ray (ray tracing), pts/gcrypt (CAMELLIA256-ECB Cipher), pts/compress-7zip (compression), and pts/himeno (poisson pressure solver). Figure 13 shows the average overhead of these benchmarks to be 4.5%, This overhead is attributed to the hardware second-stage page table coherency and inter-core TLB and caches.

Lastly, we execute a suite of I/O bound application benchmarks. We use the Phoronix pts/apache benchmark (1,000,000 requests with 100 requests being carried out concurrently), pts/fs-mark (1000 files and max. 1MB size) for SD and USB disk benchmarking, and obexftp [33] with a 32KB file transfer via bluetooth using the OBEX protocol. We also wrote a small application, gpio, to generate a wave via the GPIO pins for GPIO benchmarking. Figure 14 shows the IO benchmark results with an average overhead of 3.5%. All IO benchmarks except for pts/apache run with minimal overhead (1%). pts/apache incurs a larger overhead (11%) which is due to the nature of the TCP connections and large number of forks and exec which stress both the DMA protection as well as the memory and interrupt protection mechanisms.

7.5. Uberapps Benchmarks

We use the uberapps described in §6 for uberapp performance benchmarking. For the `ctxtrace` and `wdog` standalone uberapps, we use the `lmbench3` benchmark suite to measure the impact on process context switches and interrupt handling respectively (Figure 15). The overheads are very reasonable considering the active runtime monitoring nature of these uberapps.

For the `encfs` uberapp we use the `iozone` benchmark to measure the disk read and write performance with a target encrypted FUSE file-system container. We use iozone with 4K record size and direct IO with a 16MB file. For the `utpm` uberapp we wrote a simple user-mode test application which invokes the micro-TPM PCR read, PCR extend, seal and unseal functions. Figure 17 shows the performance of the uberapps compared to the native system without UPI. The average overhead is small (6%) which is primarily attributed to synchronous uberapp calls.

7.6. Comparative Analysis Effort

We also tried to compare UPI quantitatively with other other general-purpose hypervisors whose execution on PI seemed plausible. Our candidates were: Xen-ARM [50], KVM-ARM [19], [20], and XVisor [34].

Xen-ARM fails to build or run on the PI due to lack of virtualizable GIC and IOMMU support. This is consistent with various reports describing Xen-ARM's lack of support for the PI [2], [36], [49].

KVM-ARM built successfully, but building and getting the QEMU part to run remains a challenge — we ended

Benchmark	Native	uPI/ ctxtrace	Overhead
2p/0k ctxsw	6	7.1	1.1
2p/16k ctxsw	5.9	6.7	0.8
2p/64k ctxsw	5	6.3	1.3
8p/16k ctxsw	6.5	8.0	1.5
8p/64k ctxsw	22.9	23.3	0.4
16p/16k ctxsw	7.4	9.6	2.2
16p/64k ctxsw	24.9	29.4	4.5

Figure 15. uPI `ctxtrace` uberapp `lmbench3` process context switch benchmarks. *Values in micro-seconds, smaller-is-better.*

Benchmark	Native	uPI/ wdog	Overhead
sig inst	0.79	0.81	0.02
sig hndl	3.51	3.65	0.14
prot fault	0.50	0.53	0.03
page fault	1.42	2.09	0.67

Figure 16. uPI `wdog` uberapp `lmbench3` interrupt/signal/fault processing benchmarks. *Values in micro-seconds, smaller-is-better.*

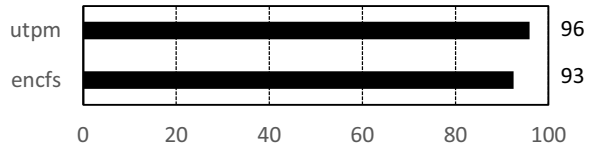

Figure 17. uPI `encfs` and `utpm` uberapps benchmark execution as % native system performance without uPI.

up with system freeze and crashes. While one report suggested the use of an out-of-tree QEMU patch and using just a single-core [39], we were unable to reproduce the results described therein. This experience is corroborated by another effort [44] which reports to have some success with a specific OpenSUSE OS image. However, this method did not work for us either.

Finally, we tried the XVisor [34] v0.2.9 monolithic hypervisor which is reported to run on the PI [37]. However, in our experiments although XVisor [34] built successfully per the instructions, it resulted in a system freeze on start-up. Further investigation revealed that others have reported similar build and runtime issues on the PI with XVisor [37], [51]. Some discussions suggest using the development release of Xvisor [52]. However, we did not have any success with that either.

8. Experience and Lessons Learned

We now describe nuances of the PI platform we discovered during the design and development process.

GPU Boot-loaders and Boot-ROM: The PI GPU is based on the VC4 architecture and can run regular applications. The first and second-stage GPU boot-loaders and examples of such VC4 applications. While the current GPU boot-loaders are binary-only, Broadcom has made the VC4 engine public and there are tools now to support development on the GPU side. There are already fledgling projects that aim to provide open-source GPU boot-loaders for the PI [4]. Further, our experiments revealed that the SoC boot-ROM can be read and dumped into a binary. The boot-loaders and binary boot-ROM can then be validated for correctness via existing source and binary verification approaches [29], [46]. We leave such exploration for future work to obtain higher assurance for the initial boot process.

Secure Boot and NVRAM: While the PI does not currently have support for secure booting in the SoC boot-ROM, there is hardware support in the form of Non-Volatile RAM (NVRAM) within the SoC, which can be leveraged to store keys and perform secure boot on the GPU side. While the NVRAM is undocumented, we were able to read and write to the NVRAM from the VC4 side in our experiments. Further, the NVRAM can be protected from the ARM side using the bus MMU. Secure boot can be added to the PI boot-ROM without any cost overhead since all the required hardware capabilities are already in place. Further, this can be done without severely impacting current development practices. One approach would be for Broadcom to provision the boot-ROM with support for signed execution of OEM (e.g., PI Foundation) boot-modules. The OEM can then support a variety of boot-up modes for the end user or system developer including both signed and unsigned kernel images.

ARM Secure-World: The PI actually allows access to the ARM secure-world (EL3). While this is undocumented, we were able to add an ARM stub on the boot-partition, which in turn can be loaded by the second-stage GPU boot-loader prior to loading `kernel.img`. This stub can then perform required platform initialization in secure-world (e.g., configuring appropriate non-secure world accesses), prior to changing the execution mode to EL2. However, there is no protected secure-world memory available precluding the use of existing secure-world architectures (§9).

ARMv8 Nuances: We now describe ARMv8 nuances which we encountered during development which were either undocumented or incorrectly documented. Memory cacheability, shareability and address sizes in the μHV core page-tables and the second-stage page-tables must match. This applies to both device (always mapped as nGRE) and normal memory regions. The PI's Cortex-A53 cores have hardware cache coherency in second-stage page-tables. Thus, there is no need for performance-impacting TLB shootdowns for second-stage page-table modifications. Lock instructions cannot execute without MMU being enabled and locks don't work on non-cacheable memory.

DMA Protection: The legacy DMA controller data-sheet does not mention the possibility of cyclic control blocks. We inferred this from the Linux kernel Broadcom driver sources after repeated system freezes during the development process.

9. Related Work

Our work uses a micro-hypervisor based system security architecture along with lightweight trap-inspect-forward to realize practical security on the low-cost PI. We discuss related work and their applicability to the PI under four

categories: ARM TrustZone [10] based architectures, OS containers, baremetal approaches and specialized add-on hardware.

TrustZone-based Architectures: One strategy for introducing security properties is to put sensitive code in the secure-world (TrustZone [10]) where it can execute in isolation. Several research proposals [21], [22], [26], [27], [30], [48], [53] employ TrustZone to achieve isolation and provide a range of security properties. However, as described in §8, although the PI supports TrustZone and has the ability to execute code in the secure-world, it lacks protected secure-world exclusive memory. Thus, although some of these approaches can be adapted to the PI, the OS and applications running in the non-secure world will have unrestricted access to the entire memory thereby undermining the integrity and secrecy of the secure-world application. Further, TrustZone lacks support for security applications that depend on runtime monitoring (e.g., `ctxtrace`).

OS Containers: OS support can be leveraged towards a general solution for isolation and resource containerization. Linux Containers [3] offer light-weight OS-level containerization by leveraging kernel namespaces. Cells [5], [18] enables device namespaces and proxies that integrate with lightweight OS virtualization to multiplex hardware across multiple virtual containers on the Android OS. KVM-ARM [19], [20] employs a hosted (type-2) hypervisor approach to run virtual machines on ARM platforms under Linux. The common drawback of hosted containers is the large TCB which includes the entire OS kernel, drivers and libraries. Cells is only geared towards mobile phones running the Android OS and does not run on the PI. Linux containers and KVM-ARM (which does not on the PI; §7.6) do not provide full access to the underlying hardware and cannot run the stock PI commodity OSes. KVM-ARM also suffers considerable performance overhead attributable to the multiple OS to hypervisor transitions owing to its type-2 architecture [34].

Baremetal Approaches: Xen-ARM [50] is the port of the Xen hypervisor to the ARM architecture. However, Xen-ARM does not run on the PI since it requires hardware capabilities such as IOMMU and virtualizable GIC which are not present on the PI. There is also performance issues due to the micro-kernel architecture and the cost of transitions between guest VMs and the driver VM [34]. XVisor [34] is a monolithic hypervisor for ARM platforms. While it has been reported to run on the PI [37], we could not reproduce and get it to run for both the latest stable and development versions (§7.6). Further, due to the monolithic nature, it has a large TCB and faces similar problems with device emulations and vulnerabilities as regular VMMs. Furthermore, XVisor does not run stock PI OSes but runs a stripped down version of Linux on an emulated platform. This precludes access to most PI peripherals from within the guest. Epoxy [16] uses a compiler based approach to isolate privileged operations on low-cost ARM embedded platforms. However, it requires full recompilation of the software stack with support for only a ARM-7M platform which makes it inapplicable to the PI platform hardware or development ecosystem.

Specialized Add-on Hardware: Add-on hardware make use of physically separated protected modules with their own processing abilities to achieve isolation and protection of sensitive information [17], [21], [42], [54]. Unfortunately, addition of such protected modules don't come cheap. For example, Zymkey costs as much as the PI with less than fourth of its processing capabilities. Further, execution within the protected module is both memory and interface constrained. For example, the aforementioned modules only allow execution of small scripts or applets with meager memory resources.

10. Conclusions and Future Work

Taking stock of the current crop of IoT/embedded computing platforms, our overarching goal was to realize practical performant security on a ubiquitous, low-cost computing platform with a low TCB, without sacrificing commodity compatibility. We found our answer in the low-cost (sub $35) Raspberry PI for which we present the first security oriented system level architecture (called UBERPI) and implementation, and conduct a comprehensive evaluation which together substantiate our goals for practical security with commodity compatibility, high-performance and low-TCB.

Future work involves leveraging the low-TCB and low complexity nature of UBERPI to perform formal verification for higher assurance as well as supporting other unmodified OSes (e.g., Windows IoT Core). We also seek to explore the application of the framework towards more real-world critical embedded and IoT landscape.

Availability

UBERPI is open-source and is available as part of the UBER-eXtensible Micro-Hypervisor Framework (UBERXMHF) at:

http://uberxmhf.uberspark.org

Acknowledgements

We thank our shepherd, Deepak Garg, for his help with the final version of this paper, as well as the anonymous reviewers for their detailed comments and feedback. This work was funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002.[8]

8. Copyright 2018 Carnegie Mellon University and IEEE. All Rights Reserved. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute. Carnegie Mellon is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University. DM18-0254

References

[1] Eben Upton: The Raspberry Pi Pioneer. IEEE Spectrum, 2015.

[2] Attempts to get Xen Hypervisor and MirageOS running on Raspberry Pi 3. https://github.com/rudenoise/xen-mirage-rpi3, 2016.

[3] Linux Containers. http://linuxcontainers.org, 2017.

[4] Open source VPU side bootloader for Raspberry Pi. https://github.com/christinaa/rpi-open-firmware, 2017.

[5] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh. Cells: A virtual mobile smartphone architecture. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 173–187, 2011.

[6] ARM. Advanced eXtensible Interface Protocol Specification. http://infocenter.arm.com, 2017.

[7] ARM. Advanced Microcontroller Bus Architecture Reference. http://infocenter.arm.com, 2017.

[8] ARM. ARM Architecture Reference Manual - ARM v8. http://infocenter.arm.com, 2017.

[9] ARM. ARM Cortex-A53 MPCore Processor - Technical Reference Manual. http://infocenter.arm.com, 2017.

[10] ARM Security Technology. Building a Secure System using Trustzone Technology. http://infocenter.arm.com, 2017.

[11] A. Awad, S. Kadry, B. Lee, and S. Zhang. Property based attestation for a secure cloud monitoring system. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, UCC '14, pages 934–940, Washington, DC, USA, 2014. IEEE Computer Society.

[12] Azema J. and Fayad G. M-Shield Mobile Security Technology: Making Wireless Secure - Texas Instrument Whitepaper, 2008.

[13] Broadcom. BCM2835 ARM Peripherals. https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2835/BCM2835-ARM-Peripherals.pdf, 2017.

[14] Broadcom. BCM2836 ARM Peripherals. https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2836/QA7_rev3.4.pdf, 2017.

[15] Broadcom. VideoCore IV 3D Architecture Reference Manual. https://docs.broadcom.com/docs-and-downloads/docs/support/videocore/VideoCoreIV-AG100-R.pdf, 2017.

[16] A. A. Clements, N. S. Almakhdhub, K. S. Saab, P. Srivastava, J. Koo, S. Bagchi, and M. Payer. Protecting bare-metal embedded systems with privilege overlays. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2017.

[17] V. Costan, L. F. Sarmenta, M. van Dijk, and S. Devadas. The trusted execution module: Commodity general-purpose trusted computing. In *Proceedings of CARDIS*, 2008.

[18] C. Dall, J. Andrus, A. Van't Hof, O. Laadan, and J. Nieh. The design, implementation, and evaluation of cells: A virtual smartphone architecture. *ACM Trans. Comput. Syst.*, 30(3):9:1–9:31, Aug. 2012.

[19] C. Dall, S.-W. Li, J. T. Lim, J. Nieh, and G. Koloventzos. Arm virtualization: Performance and architectural implications. *SIGARCH Comput. Archit. News*, 44(3):304–316, June 2016.

[20] C. Dall and J. Nieh. Kvm/arm: The design and implementation of the linux arm hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 333–348, New York, NY, USA, 2014. ACM.

[21] K. Dietrich and J. Winter. Towards customizable, application specific mobile trusted modules. In *Proceedings of the Fifth ACM Workshop on Scalable Trusted Computing*, STC '10, pages 31–40, New York, NY, USA, 2010. ACM.

[22] J.-E. Ekberg, N. Asokan, K. Kostiainen, and A. Rantala. Scheduling execution of credentials in constrained secure environments. In *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing*, pages 61–70, 2008.

[23] Emlid. Real-time preemptible kernel for Raspberry Pi. https://github.com/emlid/linux-rt-rpi, 2018.

[24] A. Fitzek, F. Achleitner, J. Winter, and D. Hein. The andix research os - arm trustzone meets industrial control systems security. In *Proceedings of IEEE 13th International Conference on Industrial Informatics (INDIN)*, 2015.

[25] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. N. Wu, S.-C. Weng, H. Zhang, and Y. Guo. Deep specifications and certified abstraction layers. In *Proc. of POPL*, 2015.

[26] J. E. Ekberg and M. Kylanpaa. Mobile Trusted Module: an introduction. Technical Report NRC-TR-2007-015, Nokia Research Center, 2007.

[27] J. E. Ekberg and M. Kylanpaa. MTM implementation on the TPM emulator. http://mtm.nrsec.com, 2008.

[28] K. K., E. J. E., and A. N. On-board credentials with open provisioning. In *Proceedings of 4th International Symposium on Information, Computer and Communications Security*, 2009.

[29] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, Feb. 2014.

[30] K. Kostiainen, J.-E. Ekberg, N. Asokan, and A. Rantala. On-board credentials with open provisioning. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, pages 104–115, 2009.

[31] Lily Hay Newman. The Botnet that broke the Internet isn't going away. https://www.wired.com/2016/12/botnet-broke-internet-isnt-going-away/, 2016.

[32] Lucian Constantin. Hackers found 47 new vulnerabilities in 23 IoT devices at DEF CON. http://www.csoonline.com/article/3119765/security/hackers-found-47-new-vulnerabilities-in-23-iot-devices-at-def-con.html, 2016.

[33] ObexFTP. Open-Source OBEX Implementation. http://dev.zuckschwerdt.org/openobex/wiki/ObexFtp, 2007.

[34] A. Patel, M. Daftedar, M. Shalan, and M. W. El-Kharashi. Embedded hypervisor xvisor: A comparative analysis. In *Proceedings of the 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 682–691, 2015.

[35] Phoronix Test Suite. Open-Source Linux Benchmarking. http://phoronix-test-suite.com, 2017.

[36] Raspberry Pi Discussion. https://raspberrypi.stackexchange.com/questions/45930/is-it-possible-to-use-any-virtualization-technique-with-the-raspberry-pi-3, 2016.

[37] Raspberry Pi Forums. Xvisor ARM hypervisor ported to Raspberry Pi. https://www.raspberrypi.org/forums/viewtopic.php?t=45081, 2015.

[38] M. Rushanan and S. Checkoway. Run-dma. In *Proceedings of USENIX Workshop on Offensive Technology (WOOT)*, 2015.

[39] Sergio L. Pascual. Enabling KVM virtualization for Raspberry Pi 2. https://blog.flexvdi.com/2015/03/17/enabling-kvm-virtualization-on-the-raspberry-pi-2/, 2015.

[40] Simon Bisson. Microsoft's Novel Approach to Securing IoT. http://www.infoworld.com/article/3193742/internet-of-things/microsofts-novel-approach-to-securing-iot.html, 2017.

[41] U. Steinberg and B. Kauer. Nova: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 209–222, New York, NY, USA, 2010. ACM.

[42] Sun Microsystems Inc. Java card specifications v3.0.1: Classic edition, 2009.

[43] Trustonic. Trusted Execution Environment. http://www.trustonic.com, 2014.

[44] Valentine Nwachukwu. Setting up KVM on Raspberry Pi 3 using a 64bit openSUSE Pi3 Leap 42.2 xfce image. https://medium.com/@valdiz777/setting-up-kvm-on-raspberry-pi-3-using-a-64bit-opensuse-pi3-leap-42-2-xfce-image-22faddf02f48, 2017.

[45] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta. Design, implementation and verification of an extensible and modular hypervisor framework. In *Proc. of 2013 IEEE Symposium on Security and Privacy*, 2013.

[46] A. Vasudevan, S. Chaki, P. Maniatis, L. Jia, and A. Datta. überspark: Enforcing verifiable object abstractions for automated compositional security analysis of a hypervisor. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 87–104, Austin, TX, 2016. USENIX Association.

[47] Q. N. G. V. P. A. Vasudevan A., Parno B. Lockdown: Towards a safe and practical architecture for security applications on commodity platforms. In *Trust and Trustworthy Computing*, 2012.

[48] J. Winter. Trusted computing building blocks for embedded linux-based arm trustzone platforms. In *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing*, STC '08, pages 21–30, New York, NY, USA, 2008. ACM.

[49] Xen Mailing List. https://lists.gt.net/xen/users/369667, 2016.

[50] Xen Team. Xen ARM with Virtualization Extensions Whitepaper. https://wiki.xen.org/wiki/Xen_ARM_with_Virtualization_Extensions_whitepaper, 2016.

[51] XVisor Mailing List. xvisor- linux does not boot raspberry pi. https://groups.google.com/forum/#!topic/xvisor-devel/D5bj6U9cs40, 2016.

[52] XVisor Mailing List. Xvisor on a Raspberry PI B+ Board. https://groups.google.com/forum/#!searchin/xvisor-devel/raspberry$20pi%7Csort:relevance/xvisor-devel/liugxJV8Vx0/MgEYXCVpAgAJ, 2017.

[53] X. Zhang, O. Accmez, and J.-P. Seifert. A trusted mobile phone reference architecturevia secure kernel. In *Proceedings of the 2007 ACM Workshop on Scalable Trusted Computing*, STC '07, pages 7–14, New York, NY, USA, 2007. ACM.

[54] Zymbit. Zymkey Key Management Card. http://zymbit.com, 2017.