Hermes: A Real Time Hypervisor for Mobile and IoT Systems

Neil Klingensmith
University of Wisconsin
naklingensmi@wisc.edu

Suman Banerjee
University of Wisconsin
suman@cs.wisc.edu

ABSTRACT

We present Hermes, a hypervisor for MMU-less microcontrollers. Hermes enables high-performance bare metal applications to coexist with RTOSes and other less time-critical software on a single CPU. We experimentally demonstrate that a real-time operating system scheduler does not always provide deterministic response times for I/O events, which can cause real-time workloads to be unschedulable. Hermes solves this problem by adding a layer of abstraction between the hardware I/O devices and the software that services them, making I/O transactions truly deterministic. Virtualization on low-power mobile and embedded systems also enables some interesting software capabilities like secure execution of third-party apps, software integrity attestation, and bare metal performance in a multitasking software environment. These features otherwise require additional hardware (i.e. multiple CPUs, hardware TPM, etc) or may not be available at all. In other projects, we have anecdotally noticed that real time operating systems are not always able to respond quickly and deterministically enough to time-sensitive operations, particularly under high I/O load. We validate this observed timing problem by measuring interrupt latency in an RTOS environment and comparing to an experimental implementation of Hermes. We find that not only is the interrupt latency lower in the virtualized environment, but it is also much more deterministic—a key figure of merit for real-time software systems. We discuss challenges of implementing a hypervisor on a CPU with no memory management unit, and we present some preliminary solutions and workarounds. We go on to explore some other applications of virtualization to mobile and IoT software.

CCS CONCEPTS

• **Software and its engineering** → *Virtual machines*; • **Computer systems organization** → **Real-time operating systems**; **Embedded systems**;

KEYWORDS

Real-time systems; hypervisor; virtualization

ACM Reference format:
Neil Klingensmith and Suman Banerjee. 2018. Hermes: A Real Time Hypervisor for Mobile and IoT Systems. In *Proceedings of 19th International Workshop on Mobile Computing Systems & Applications, Tempe , AZ, USA, February 12–13, 2018 (HotMobile '18),* 6 pages.
https://doi.org/10.1145/3177102.3177103

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HotMobile '18, February 12–13, 2018, Tempe , AZ, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5630-5/18/02…$15.00
https://doi.org/10.1145/3177102.3177103

1 INTRODUCTION

Modern embedded sensing and mobile applications increasingly perform diverse functions, including displaying user interfaces, managing networking, performing real-time data acquisition, and more. Some even allow third-party code to be downloaded and run alongside the factory firmware [17]. Such diversity in runtime requirements poses challenges to software architects, who must manage the often competing needs of different tasks.

To manage the diverse runtime requirements of embedded software, we have developed a lightweight embedded hypervisor we call Hermes[1], targeted to ARM Cortex-M microcontrollers. Other authors have proposed similar systems for mobile phone environments, but none that we are aware of on MMU-less processors [5, 9, 14].

IoT applications are frequently implemented on CPUs without an MMU in order to save cost and power. While the cost of MMU-ful Linux-capable processors is going down all the time, energy considerations (especially for mobile applications) are not likely to go away.

The problem we set out to solve is one of I/O latency in such a complex runtime environment. Real-time operating system (RTOS) scheduling algorithms cannot guarantee deadlines will be met under high I/O load. People usually solve this problem by running time-critical operations on a separate CPU [13]. For example, high-frequency signal sampling may be implemented in bare metal code running on an independent microcontroller while the user interface, networking, storage, etc. runs on the main device. This approach has a lot of obvious shortcomings: increased hardware and software complexity, power consumption, physical size, verification difficulty, etc.

Driver-level I/O processing has traditionally been assumed to be a negligible component of overall response time—an assumption that was valid 30 years ago as these real-time scheduling algorithms were being developed. At that time, embedded computers were single-purpose machines that largely performed the same task repetitively.

But that assumption of single-purposeness is becoming less valid. Modern microcontrollers are equipped with a diverse range of peripherals that was unimaginable in the 1980s.Network interfaces, high-speed data acquisition devices, touchscreens, and more all have a diverse range of requirements, but they are treated the same by the RTOS and CPU. Exception management for low-priority I/O

[1] **H**yp**E**rvisor for **R**eal time **M**icrocontroll**E**r**S**
`http://hermes.wings.cs.wisc.edu`

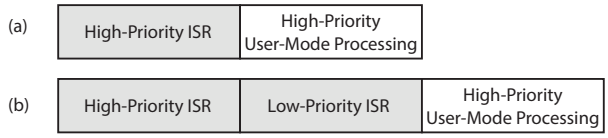

Figure 1: Timeline of (a) high-priority ISR followed by user-mode I/O processing and (b) low-priority ISR co-occuring with a high-priority ISR, delaying high-priority user mode processing.

Software Environment	Entropy of Latency
FreeRTOS, Serial Only	0.85
FreeRTOS, Serial + Ping Flood	1.78
Bare Metal Guest, Serial Only	0.27
Bare Metal Guest, Serial + Ping Flood	0.14

Table 1: Entropy of the distributions of latency measurements (distributions shown in Figure 2). Low values of entropy are more deterministic. The bare metal guest running in Hermes has much more predictable latency than tasks in FreeRTOS. Under Hermes, latency is still highly deterministic under high I/O load.

is always performed before user-mode code can respond to high-priority events, creating a kind of unintended priority inversion (depicted in Figure 1). Consequently, response times to latency-sensitive I/O events are not deterministic, which can result in failure (see an exploration of this in Section 2).

Conventional wisdom among real-time programmers is that ISRs should be as short as possible: clear the interrupt, maybe transfer a few bytes of data, and exit. Userland code should be responsible for responding to the event. In a crowded software environment with multiple drivers and tasks competing for CPU time, this programming method has the effect of delaying the actual response of all I/O events until all ISRs have finished executing. These delays break the assumptions that underlie real-time scheduling algorithms, which require the highest-priority task to always run first. Instead, we are running the driver code associated with low-priority tasks before the user code for high-priority tasks, and RTOSes do not have flexibility to change this behavior.

Hermes is a lightweight virtualization platform that lives between the hardware and the operating system. At its core, Hermes consists of some initialization code and a single interrupt service routine that catches and preprocesses all exceptions before dispatching them to the operating system. In its role as a mediator of exception processing, Hermes can allow multiple operating systems or bare metal applications to run side-by-side on an MMUless microcontroller, dispatching exception processing to the appropriate OS as necessary, much like a hypervisor running on a PC or server. We see several potential advantages to this software architecture:

(1) **Performance.** For time-critical applications, Hermes can provide a thin layer between the software and the hardware. Real time operating systems (RTOSs) on the other hand, often come with a lot of overhead in the form of system call latency for time-critical tasks. This may be unacceptable in applications where time-critical tasks need to coexist with other less critical code like networking or user interface software.
(2) **Security.** With Hermes, we can allow untrusted third party code to run in a sandboxed environment that protects mission critical software from attacks. Hermes, with the assistance of peripherals commonly available on commodity microcontrollers, can provide a root of trust for guests and remote agents.
(3) **Portability.** Hermes can provide a consistent virtual environment for all higher level software, regardless of the underlying hardware. This could enable, for example, edge computing devices with heterogeneous hardware implementations to run user apps targeted to a common platform.

A diagram of the Hermes software architecture is shown in Figure 3. We are implementing Hermes on an ARM Cortex M7 CPU called the Atmel SAM E70 [6, 7] which has 2 Mbytes of flash and 384 kbytes of RAM. It also includes many advanced features of the latest ARM microcontrollers such as a floating point unit, a memory protection unit, separate instruction and data caches, and many peripherals. We have tested Hermes by running a FreeRTOS v9.0.0 [2] guest on top of the hypervisor. The contributions made by this work are the following:

- We measure the response time to interrupts in an RTOS environment, demonstrating that latencies can be nondeterministic under high I/O load.
- We propose the use of a hypervisor in real time software environments to ameliorate the determinism problem. The hypervisor can provide isolation between software tasks, which can improve timing predictability.
- We develop a rudimentary implementation of the hypervisor, and we study its performance. We discuss some of the difficulties of implementing a hypervisor on a microcontroller with no MMU.

2 PROBLEM VALIDATION

Using performance counters on the ARM Cortex M7 CPU, we measure the ISR-user space latency—the time between beginning of ISR execution to beginning of userspace data processing. This is a metric of how long it takes to respond to an I/O event. Ideally, for time sensitive I/O this time should be short and deterministic, meaning the same for each I/O event. We find that in the FreeRTOS environment, the ISR-user space latency is **less deterministic** under high I/O load, as expected. We have also experienced this problem when developing other systems, but we did not study it as carefully [12].

Experimental Setup

We measured the ISR-userspace latency for a serial port receive in FreeRTOS and Hermes. In FreeRTOS, we used an OS queue to transfer the data from an ISR to a user-mode task. In Hermes, we ran a FreeRTOS guest alongside a bare metal guest that transferred data between an ISR and userspace code using a memory buffer. In both runtime environments, we had two other periodic FreeRTOS tasks running alongside the latency test. In both cases, we ran the latency

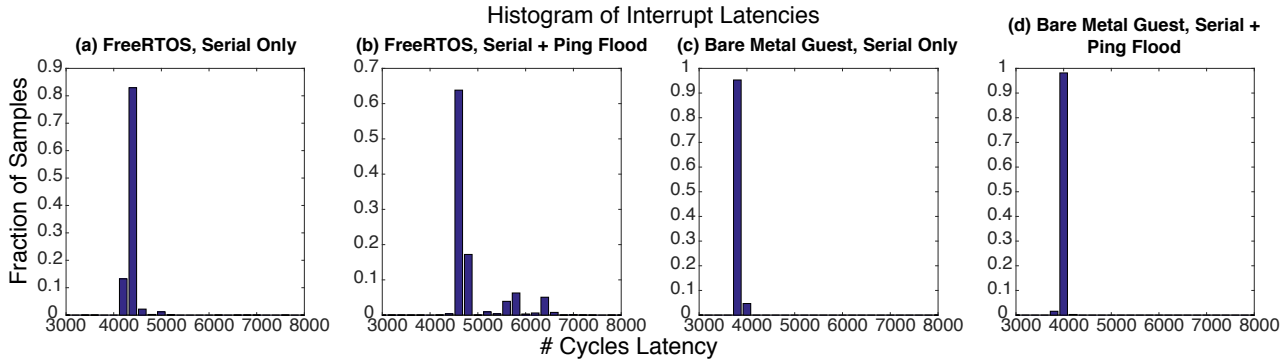

Figure 2: ISR-userspace latency histograms. Latency is measured as the number of cycles elapsed between executing the serial port receive ISR and beginning of userspace processing.

test in isolation as well as in the presence of high I/O load (a ping flood) to test how well each software environment could provide a deterministic runtime environment. The networking software that responded to the pings was implemented as a low-priority task in FreeRTOS for both environments.

FreeRTOS

FreeRTOS is a popular (if not the most popular) real-time operating system for embedded and IoT computers. It has been ported to CPUs manufactured by 20+ manufacturers representing every commonly used architecture (ARM, x86, etc.). FreeRTOS implements a rate monotonic scheduler in which each task has a fixed priority, and the highest priority task that is ready to run is executed first.

According to its development team, "FreeRTOS never performs a non-deterministic operation, such as walking a linked list, from inside a critical section or interrupt." Traditionally, its set of features—deterministic rate monotonic scheduler—is thought to provide deterministic event response times. This is true for CPU-intensive tasks, but, as we will see, that assumption of determinism breaks down under high I/O load.

Results

Figure 2 shows the results of our latency tests. Each subplot is a histogram of ISR-userspace latencies. Ideally, we would want these plots to have only one bar—a single response time for every I/O event. Figure 2 (a) and (b) show latency in FreeRTOS only, under low and high I/O load respectively. Under high I/O load, the latency histogram is more spread out because exceptions raised by unrelated I/O events delay execution of the user mode code in an unpredictable way. This happens when a serial port exception and a network port exception occur close in time. Both exceptions must be processed before the user mode code to handle the serial port receive can begin executing. We get shorter and more deterministic response times when the serial port exception occurs in isolation. If the network port exception occurs near the same time as the serial port exception, the network port ISR will have to execute before the CPU can return to user mode, delaying the response time. This is an inherent disadvantage of running multiple unrelated programs on a single processor which we are trying to correct with Hermes.

Figure 2 (c) and (d) show the latency of the same I/O operation running as a bare-metal guest inside Hermes. Determinism is higher for histograms that are more clustered around a single value and lower for histograms that are more spread out.

Discussion

The reason that ISR-userspace latency is more deterministic in Hermes under high I/O load is that by design, Hermes can enable or disable different interrupt sources depending on which guest is active. In this test, we disabled the network port exception when the bare-metal serial port guest was running. This makes it impossible for the network port ISR to interrupt the user-mode code that handles the serial port receive. Operating systems in general do not support changing processor state for different threads[2], presumably because I/O transactions are assumed to be the domain of the operating system and mostly independent of user-mode software. That assumption was generally valid for early PCs and servers, whose job was primarily batch-mode processing with very little user interaction. Mobile and IoT devices have completely different set of requirements: they need to serve as a responsive user interface in which software works closely with I/O.

Uncertainty in scheduling can create real problems for these kinds of systems. For instance, if the same timing uncertainty in Figure 2 were imposed on ADC sampling in an IoT device, it could cause several decibels of harmonic distortion [3]. It is easy to imagine many situations in which timing errors could result in degraded system performance on mobile platforms.

We should be clear here that we do not have a full implementation of I/O prioritization in Hermes. The results in Figure 2 were obtained by forcing a context switch to the bare metal guest each time we took a serial port exception. We acknowledge that our implementation is not scalable, but the results give us some insight into what is possible with virtualization. Cleaning up the implementation should not significantly degrade performance of the hypervisor.

We considered several simpler alternative solutions that could be implemented in the RTOS to alleviate I/O latency:

(1) **Disable interrupts in user-level code.** This would allow us to process the I/O event in userspace without interruption. It does not solve the problem of ISR-userspace latency, since

[2]For example, we are not aware of any RTOS that allows the programmer to enable or disable different drivers while certain threads are running.

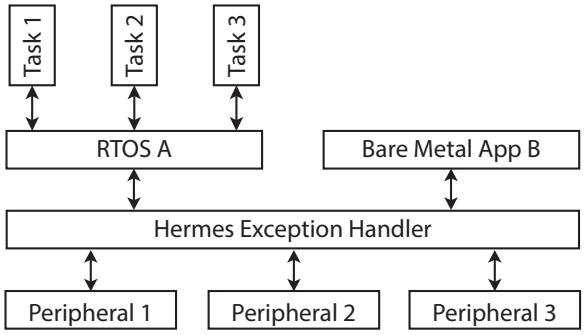

Figure 3: Architecture of the Hermes Hypervisor. The main component of Hermes, its monolithic exception handler, intercepts all exceptions before dispatching them to the guests.

more than one exception may execute sequentially before user code gets a chance to disable interrupts.

(2) **Process I/O events in the ISR.** This would allow us to ensure that our I/O events are processed in a timely fashion. This could be an acceptable solution for a single-purpose bare-metal app with no other tasks running concurrently. The problem with this approach in an RTOS is that it monopolizes the CPU during the entire I/O operation, likely causing other tasks to hang while the I/O event is handled.

(3) **Re-prioritize the interrupts.** We could use the CPU's interrupt prioritization circuits to execute the time-critical ISR first, before other ISRs. This wouldn't decrease latency in an RTOS environment because lower priority ISRs will always execute before the user space code.

None of these solutions is a viable alternative because they cannot reduce latency while maintaining a responsive runtime environment for other concurrent tasks.

3 ARCHITECTURE

In its current implementation, Hermes is a single monolithic interrupt service routine that intercepts all CPU exceptions before they can be processed by the operating system. Figure 3 shows a diagram of the interactions between the Hermes hypervisor and its guests. On boot, the Hermes initialization code sets up the CPU's exception table to point to the Hermes ISR. It then launches the guest operating systems in the ARM CPU's unprivileged execution mode[3].

3.1 Opportunities

Running a hypervisor on embedded IoT equipment enables some interesting possibilities for IoT software.

Distributed Processing on a Single Chip

Many embedded hardware designs use a distributed computation model to separate a complex task into several independent execution environments. For example, a board might have one network processor, one sampling processor, and a main CPU, each performing its own specific task independently of the others. This type of design complicates the hardware and software and likely drives up the cost, size, and energy requirements of the equipment. With a hypervisor, we can run all software on a single CPU while maintaining isolation by running each independent application in its own VM. CPU and resource allocation can be strictly controlled by the hypervisor to ensure that deadlines are met. Our preliminary investigation into I/O latency in Section 2 suggests that we can use Hermes to isolate low-priority CPU time hogs (like the ping flood) from the rest of the system.

Security

Authenticating the software on an unattended embedded device is still an open problem. A few proposed solutions [4, 15] rely on measuring the timing of some arbitrary computational operation. The hypervisor may be able to serve as a root of trust for virtualized applications by implementing a virtualized trusted platform module (vTPM) [1] to be used by underlying software components. It may be possible to implement a virtual TPM in software using either ARM TrustZone [16] or an on-chip cryptographic accelerator [10].

3.2 Challenges

Implementing a virtualization environment on a platform with no hardware support has a unique set of challenges that we will discuss here.

Compile-Time Guest Setup

Since we are dealing with a system that has no MMU, we are required to compile all guests with the hypervisor into a single runtime binary. The practical challenge is that, for symmetric guests (more than one instance of a single guest OS), we must change the name of each function and variable in order to avoid linker errors. This can be mildly annoying because it makes the RTOS code harder to read. We have written a script to perform this task automatically.

Imprecise Bus Fault Exceptions

The ARM Cortex M line of CPUs throws bus fault exceptions for accesses to privileged memory regions that are mapped to certain control registers. Some of these exceptions can be imprecise, meaning that the CPU does not record the exact instruction that caused the exception. Instead, in response to an imprecise exception, the CPU will throw a bus fault as soon as possible (in our experience 2-10 instructions past the faulting instruction). This makes the job of the Hermes exception handler difficult since it does not know which privileged memory access needs to be emulated. The only thing we can be sure of is that the faulting instruction occurs earlier in the instruction stream than where the exception was thrown.

Fortunately for us, the ARM Cortex M7 device on which we implemented Hermes sometimes records the effective address of the instruction that caused the bus fault, even if the bus fault is imprecise[4].

We solve the problem of imprecise bus fault exceptions by tracing back through the instruction stream to look for a privileged instruction with the correct effective address that is likely to have caused the imprecise exception. Starting at the address of the instruction

[3]Normally, operating system code would run in privileged execution mode, but when the RTOS is running as a guest inside Hermes, it executes in unprivileged mode.

[4]Contrary to this fact, the ARM documentation indicates that the CPU *never* records the effective address of instructions that cause imprecise bus faults. We have either found an error in the documentation or a bug in the CPU, but this is good since it works to our advantage.

that caused the exception, we trace back through the last five instructions in order of memory address. We decode each instruction and compare its effective address to the address that caused the bus fault. If the instruction's effective address matches the offending effective address, then we assume a match and emulate that instruction.

Clearly, there are some pathological cases that could cause this approach to fail. So far, we have not encountered any code in a guest that causes this approach to fail to emulate the guest.

Some Special Register Accesses Don't Cause Exceptions

Two instructions on the ARM Cortex M7—`mrs` and `msr`—which write and read certain special-purpose registers in the ARM CPU do not cause privilege violation exceptions when executed by the guest. These instructions allow access to special CPU registers that control interrupt priority masking and accesses to the master stack pointer. The `mrs` and `msr` instructions are classified as privileged instructions, but when they are executed by code running in unprivileged mode, they fail silently: the register write is not committed, and the processor continues normal execution.

The problem is that if a guest OS tries to modify the processor state with one of these instructions, that state modification cannot be registered by Hermes since it does not cause an exception. The privileged instruction will complete like a `nop` instruction without modifying the CPU state. Critical CPU state changes like disabling interrupts will not work as intended.

We circumvent this problem by patching the OS kernel, adding an undefined instruction immediately following an `mrs` or `msr`. When the hypervisor encounters an undefined instruction exception, it will search backward in the instruction stream for an `mrs` or `msr` instruction. If we run an unpatched kernel inside the hypervisor, it will crash because the intended CPU state modifications will not happen as intended.

4 I/O VIRTUALIZATION

The main goal of the Hermes hypervisor is to provide a thinner layer between hardware and software than is possible with an RTOS. There are three general techniques for virtualizing I/O:

- **Passthrough** uses interrupt and DMA remapping to give guests direct access to hardware resources.
- **Partial emulation** implements a reduced-function virtual hardware device with a custom device driver for the guest.
- **Full emulation** implements full emulation of the physical hardware device, including the full complement of registers, FIFOs, etc available on the hardware.

In this work, we studied passthrough and partial emulation, using the network interface as the target I/O device. The network driver is convenient because it is easy to benchmark using ICMP echoes (pings), and it's easy to compare to other virtualization platforms.

Figure 4 shows a comparison of round trip times for three different Ethernet driver implementations. The bare metal implementation is the unmodified driver supplied by the chip manufacturer with no virtualization; it is our reference implementation.

The bridged implementation is a custom driver running in the guest. Ethernet device interrupts are handled by Hermes without being passed up to the guest. The hypervisor presents a virtualized network interface to the guest, and they hypervisor calls the chip

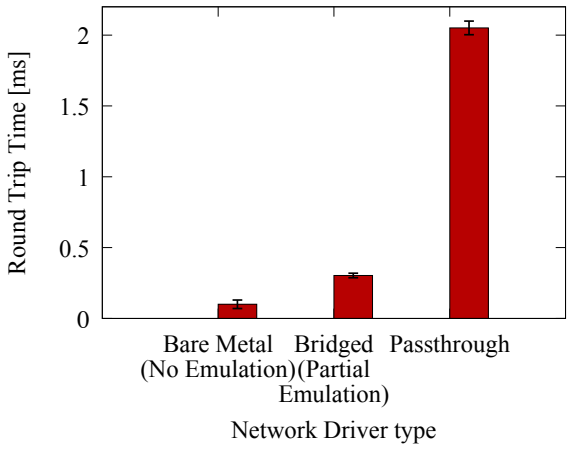

Figure 4: Comparison of ping round trip times for three Ethernet driver implementations on the ARM device.

manufacturer's driver functions to send and receive packets. The bridged driver allows multiple guests to share the same network interface by multiplexing incoming packets to the guests based on MAC address.

In the passthrough implementation, the guest runs the manufacturer's driver in raw form, emulated by Hermes. Ethernet device interrupts are caught by Hermes and passed to the guest, so all exception handling code is done in guest mode. The Ethernet device is not shared among multiple guests in this configuration.

Surprisingly, we find that the bridged (hypervisor-assisted) Ethernet driver performs far better than the passthrough. Since the passthrough driver runs all driver code in guest mode, all privileged instructions must be emulated by Hermes. This causes a significant slowdown in packet handling because the Ethernet driver has to invalidate a lot of data cache lines each time a packet is received, which requires many privileged instructions and memory accesses. In the bridged driver, the majority of privileged memory accesses and privileged instructions are done by the hypervisor, so they don't need to be emulated.

5 RELATED WORK

Other authors have explored real-time schedulers in hypervisors, in particular for Linux running in Xen [11, 18, 19]. None that we know of have been implemented on MMUless machines—even early hypervisors ran on machines with memory management units [8].

6 ACKNOWLEDGEMENTS

All authors are supported in part by the following grants from the US National Science Foundation: CNS-1345293, CNS-14055667, CNS-1525586, CNS-1555426, CNS-1629833, CNS-1647152, CNS-1719336.

REFERENCES

[1] Stefan Berger, Ramón Cáceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. 2006. vTPM: Virtualizing the Trusted Platform Module. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume*

15 (USENIX-SS'06). USENIX Association, Berkeley, CA, USA, Article 21. http://dl.acm.org/citation.cfm?id=1267336.1267357

[2] Richard Berry. 2017. FreeRTOS. (2017). http://www.freertos.org.

[3] Brad Brannon and Allen Barlow. 2006. Aperture uncertainty and ADC system performance. *Application Note AN501* (2006).

[4] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. 2009. On the Difficulty of Software-based Attestation of Embedded Devices. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS '09)*. ACM, New York, NY, USA, 400–409. https://doi.org/10.1145/1653662.1653711

[5] Yeongpil Cho, Junbum Shin, Donghyun Kwon, MyungJoo Ham, Yuna Kim, and Yunheung Paek. 2016. Hardware-Assisted On-Demand Hypervisor Activation for Efficient Security Critical Code Execution on Mobile Devices. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 565–578. https://www.usenix.org/conference/atc16/technical-sessions/presentation/cho

[6] Atmel Corporation. 2017. SAM E ARM Cortex-M7 Microcontrollers. (2017). http://www.atmel.com/products/microcontrollers/arm/sam-e.aspx.

[7] Atmel Corporation. 2017. SAM E70 Xplained Evaluation Kit. (2017). http://www.atmel.com/tools/atsame70-xpld.aspx.

[8] R. J. Creasy. 1981. The Origin of the VM/370 Time-sharing System. *IBM J. Res. Dev.* 25, 5 (Sept. 1981), 483–490. https://doi.org/10.1147/rd.255.0483

[9] Christoffer Dall and Jason Nieh. 2014. 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)*. ACM, New York, NY, USA, 333–348. https://doi.org/10.1145/2541940.2541946

[10] Joan G. Dyer, Mark Lindemann, Ronald Perez, Reiner Sailer, Leendert van Doorn, Sean W. Smith, and Steve Weingart. 2001. Building the IBM 4758 Secure Coprocessor. *Computer* 34, 10 (Oct. 2001), 57–66. https://doi.org/10.1109/2.955100

[11] Marisol GarcÃČÂŋa-Valls, Tommaso Cucinotta, and Chenyang Lu. 2014. Challenges in real-time virtualization and predictable cloud computing. *Journal of Systems Architecture* 60, 9 (2014), 726 – 740. https://doi.org/10.1016/j.sysarc.2014.07.004

[12] Neil Klingensmith, Dale Willis, and Suman Banerjee. 2013. A Distributed Energy Monitoring and Analytics Platform and Its Use Cases. In *Proceedings of the 5th ACM Workshop on Embedded Systems For Energy-Efficient Buildings (BuildSys'13)*. ACM, New York, NY, USA, Article 36, 2 pages. https://doi.org/10.1145/2528282.2534156

[13] Fabien Le Mentec. 2014. Using the Beaglebone PRU to achieve realtime at low cost. *Embedded Related* (April 2014). https://www.embeddedrelated.com/showarticle/586.php.

[14] Carlos Moratelli, Sergio Johann, and Fabiano Hessel. 2016. Exploring Embedded Systems Virtualization Using MIPS Virtualization Module. In *Proceedings of the ACM International Conference on Computing Frontiers (CF '16)*. ACM, New York, NY, USA, 214–221. https://doi.org/10.1145/2903150.2903179

[15] Bryan Parno, Jonathan M McCune, and Adrian Perrig. 2010. Bootstrapping trust in commodity computers. In *Security and privacy (SP), 2010 IEEE symposium on*. IEEE, 414–429.

[16] Himanshu Raj, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Loeser, Dennis Mattoon, Magnus Nystrom, David Robinson, Rob Spiger, Stefan Thom, and David Wooten. 2016. fTPM: A Software-Only Implementation of a TPM Chip. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 841–856. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/raj

[17] Dale F. Willis, Arkodeb Dasgupta, and Suman Banerjee. 2014. ParaDrop: A Multi-tenant Platform for Dynamically Installed Third Party Services on Home Gateways. In *Proceedings of the 2014 ACM SIGCOMM Workshop on Distributed Cloud Computing (DCC '14)*. ACM, New York, NY, USA, 43–44. https://doi.org/10.1145/2627566.2627583

[18] Sisu Xi, Chong Li, Chenyang Lu, Christopher D Gill, Meng Xu, Linh TX Phan, Insup Lee, and Oleg Sokolsky. 2015. RT-Open Stack: CPU Resource Management for Real-Time Cloud Computing. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*. IEEE, 179–186.

[19] Sisu Xi, Meng Xu, Chenyang Lu, Linh TX Phan, Christopher Gill, Oleg Sokolsky, and Insup Lee. 2014. Real-time multi-core virtual machine scheduling in xen. In *Embedded Software (EMSOFT), 2014 International Conference on*. IEEE, 1–10.