Securing DMA through Virtualization

Oliver Schwarz
Swedish Institute of Computer Science
Email: oliver@sics.se

Christian Gehrmann
Swedish Institute of Computer Science
Email: chrisg@sics.se

***Abstract*—We present a solution for preventing guests in a virtualized system from using direct memory access (DMA) to access memory regions of other guests. The principles we suggest, and that we also have implemented, are purely based on software and standard hardware. No additional virtualization hardware such as an I/O Memory Management Unit (IOMMU) is needed. Instead, the protection of the DMA controller is realized with means of a common ARM MMU only. Overhead occurs only in pre- and postprocessing of DMA transfers and is limited to a few microseconds. The solution was designed with focus on security and the abstract concept of the approach was formally verified.**

I. Introduction

Different types of embedded systems are present in all kinds of computing and control devices ranging from low power sensor nodes to mobile phones. Not only the usage of embedded systems is increasing, also the number of their vulnerabilities is. The trend towards the usage of common software components, highly interconnected systems and open architectures increases the risk of software attacks even more, evoking the need for more powerful protection mechanisms. A promising approach is the usage of virtualization as mean to isolate security-critical from non-security-critical execution, for example a commodity operating system from trusted services. Such a separation can be achieved independently from the complexity of the guest systems. In fact, the trusted computing base can be reduced by the introduction of a virtualization layer. The smaller the trusted computing base is, the earlier it is possible to get assurance on the security achieved, for example through formal verification. The idea of using virtualization as an enabler for security is not new; however, there is still a deficit of solutions targeting the special needs of embedded systems while keeping a small code base. In [1] we presented such an approach. It targets the widespread ARMv5 platforms and focuses on security by providing isolation between guests. With a footprint of less than 10 KB the described hypervisor is reasonable thin. Furthermore, the performance overhead has been shown to be low, even though ARMv5 does not provide any advanced hardware support for virtualization. This shortcoming is expected to change in the future and the recent ARMv7 Cortex-A15 has additional hardware support to handle virtual to physical address translation [2]. Their System MMU [3] even targets *Direct Memory Access (DMA)* in particular. However, legacy embedded systems will exist for a very long time and currently there is still a lack of essential hardware functions such as assisted handling of peripheral units by an IOMMU [4].

To provide peripheral devices with quick access to the memory, DMA became essential even in embedded systems. DMA controllers are able to copy data faster than the CPU. However, most of the related work that aims at virtualization of DMA does not (or not sufficiently) take care of security matters in respect to DMA. The purpose of our paper is thus to address this issue. That is indeed needed as DMA potentially opens an alternative way for attackers to modify otherwise inaccessible memory by bypassing CPU and MMU. We show how to extend the previous hypervisor design by adding protection against such attacks. Common solutions use to address this challenge by the usage of an IOMMU. However, many embedded systems lack this hardware feature. We therefore show how to prevent DMA attacks with high secure isolation guarantees through DMA virtualization based on standard ARM *Memory Management Unit (MMU)* functionality only. Our design approach is generic and applies to newer ARM architectures such as ARMv6 and ARMv7 as well. Embedded platforms without hardware virtualization support like ARMv5 will remain widespread for many years. Especially low end devices still use early designs.

The following contributions are made in the paper:

- We describe how virtualization can be used to protect DMA on ARMv5 based systems.
- We show that secure DMA virtualization is feasible without additional hardware.
- We make a careful security and performance analysis of our design proposal.
- We provide a formal proof of the abstract concept of the approach.

The remainder of this paper is organized as follows: First, we give an overview of related work. Next, assumptions, the threat model and security requirements are defined. After an introduction to the ARM architecture and the hypervisor, the DMA virtualization approach is explained in detail. An

analysis of the results and a discussion of the concept's formal verification subsequently underline the value of the approach. Finally, we conclude the paper.

II. Related Work

Virtualization as mean to provide security in embedded systems was discussed in [5]. Previous attempts to actual use virtualization on embedded systems have mostly been focused on porting of widely used virtualization layers such as Xen [6], [7] or on performance analysis aspects [8].

That strong isolation can be achieved through advanced combination of virtualization and the standard memory protection support on for example Intel based systems was convincingly showed in the recent works by Seshadri et al in [9] and [10]. Our design allows even the support of DMA without loss of isolation or the need for advanced I/O memory protection (such as an IOMMU). Härtig et al. [11] describe how to monitor potentially malicious device drivers of a system without a hardware IOMMU. However, different from our approach, their work is neither designed to support several guests nor does it cover scheduling or virtual views on the DMA controller. Furthermore we put the assurance of security properties into the focus.

Recent work has shown that verification of low level software is in fact feasible. Klein et al. successfully verified the micro kernel seL4 [12] and the Robin project [13] made progress on the verification of the Nova hypervisor [14]. This gives us motivation to perform verification effort on our solutions (such as the here described approach) as well.

III. Prerequisites

A. Assumptions

The following assumptions are made in this paper:

- The only DMA controller present on the platform is a general purpose DMAC not bounded to any particular peripheral. It does not perform any action without being programmed to do so by the CPU. The programming interface of the DMAC is known to the hypervisor.
- The MMU supports the management of ARM domains (see Sect. IV-B). Peripherals are memory mapped and access to them can be controlled through the MMU.
- All hardware entities, including the DMAC and the MMU, are working correctly and are not malicious. No physical attacks or tampering attempts are present.
- The hypervisor code is loaded unmodified at system startup. All upstream entities (such as the BIOS and the boot loader) are trusted. To ensure this, a secure boot process [15] can be used.

B. Threat Model

The attacker is assumed to have complete control over one or several guests. That includes the possibility to execute arbitrary code and to access data with its/their rights. The attacker's goal is to compromise any other guest, that is, to modify or read its code or data, or to prevent, delay or control execution of foreign code. We assume that he or she has no intention to determine whether DMA is used by other parties or not. However, should a stricter security property be desired, the hypervisor could be extended in one of the following ways to prevent revealing delays:

- "Secret" DMA tasks can be stopped in favor of DMA tasks issued by a guest which might possibly be interested in observing the DMA activities of the system. They can be resend to the DMAC after other tasks are done.
- The hypervisor can introduce pseudo delays if the DMAC is not used by other guests, so that an attacker cannot distinguish between free and occupied DMAC channels.
- The concurrent execution of guests can be (temporarily) disallowed or the hypervisor can wait until all "secret" DMA tasks are finished before a potentially malicious guest is called.

All those measures can be provided optionally so that they only apply for certain critical guest configurations.

C. Security Requirements

The goal is to provide isolation between guests to prevent any of the attacks described in the threat model. Furthermore it shall be guaranteed that DMA functionality is always available to non-malicious guests. This means that any policy-conform DMA task will eventually be processed. Given those targets, the following security requirements need to be implemented:

1) Every attempt to access the DMA controller leads to a trap into the hypervisor.
2) The DMAC performs only those copy operations which comply with the access policy. A word is accessible according to that policy if and only if it can be written/read by the current guest even without DMA support, that is, via the CPU, but in context of the valid MMU settings.
3) No guest can (re-)program a DMA request on behalf of another guest. This is independent from the access policy. Assume guest 1 has access to the addresses A, B, C, D and intents a copying from A to B, then guest 2 is not able to alter this request to, e.g. "from C to D", even though this would be a valid request when issued by guest 1 without interference of guest 2.
4) The scheduling does not influence the security.
5) Virtualization solutions which are supposed to provide security may have the vulnerability that they are attackable in their configuration [16]. In contrast, our hypervisor can not be modified by guests, neither using DMA functionality nor otherwise.
6) All DMA tasks that comply with the policy are either directly processed by the DMAC or enqueued.
7) Every DMA request will be processed (either executed or denied) and subsequently deleted from the DMAC and internal structures, so that they can not get blocked.

The isolation target is covered by requirements 1 - 5. DMA functionality can only be used through the hypervisor (1), which itself is not modifiable (5) and works correctly. Working correctly here means that it only grants valid accesses (2) issued by the right guest (3) and that itself will not modify

accesses so that they would influence security (4). Availability follows directly from requirements 6 and 7.

IV. Architecture and Hypervisor

A. Operation Modes

ARMv5 has one non-privileged mode and six privileged modes without further hierarchy. As a hypervisor needs to supervise guest kernels, it has to operate in the privileged ring while all guests have to be placed completely into the non-privileged mode. On interrupts and data aborts, the CPU switches to the privileged ring. By using a software interrupt, guests can intentionally pass control to the hypervisor.

B. Memory Access Control

The MMU is used to ensure separation between guests and to protect the hypervisor. The ARM architecture allows to define so called "domains". Each page can be assigned to one of them. Depending on the access bits for a domain, all its pages are either not accessible at all, fully accessible or subject to the settings on page table level. This allows the simultaneous changing of memory access rights for all domains by only one register access. Input and output devices are memory mapped and are thus also subject to the MMU based access control. This enables a hypervisor to intercept on interaction with the DMAC via an according abort handler.

C. OVP and the DMA Controller

We have developed a proof of concept implementation, which has been tested on an emulated platform by *Open Virtual Platforms (OVP)* [17]. Our implementation utilizes the general purpose DMAC provided by OVP. Different from the ARM provided *PrimeCell DMA Controller (PL080)* [18], the OVP DMAC has only two channels instead of eight and is restricted to the simplest functionality. However, the programming interface is similar and the simplified DMAC is still sufficient to demonstrate the effectiveness of the approach described in this paper. The focus lies on four channel specific registers, namely a source and a destination register, the control register, in which information about the burst size and the total size of the data is encoded, and the configuration register, which causes the copying to start on certain values.

D. The Hypervisor

The starting point for our design was a hypervisor previously designed and implemented by the Swedish Institute of Computer Science. The hypervisor aims at the isolation of guests. Each guest is made up of an arbitrary number of guest modes. Besides separating different guests from each other, the hypervisor can also strengthen isolation between the guest modes of one single guest (such as the user and the kernel mode of an operating system). In either case the hypervisor can be configured to allow inter-mode-access mono- or bi-directional or to prohibit it so that isolation is guaranteed. A typical scenario would include a setup of one commodity operating system along with a domain (guest mode) for trusted services. Several memory regions can be defined, for example one for each guest mode, and pages can be assigned accordingly to determined ARM memory domains, which allow quick locking and unlocking of the guests' data and executable code. Additionally, when switching between two guest modes, the hypervisor saves and restores the respective contexts, that is, the registers' contents. For the communication between guest modes the hypervisor offers the possibility of establishing remote procedure calls (RPC). The operating system FreeRTOS was successfully ported to the hypervisor.

V. DMA Virtualization

Common to other hardware virtualization solutions, we use an approach where we emulate the DMAC. Different from typical approaches though, our design is driven by a careful security analysis, making sure that a hostile guest will not be able to circumvent any system access rules. This includes interrupt handling and memory access control. Guests do not interact directly with the physical controller. Instead, each access attempt will result in trapping into the hypervisor, which then controls and manages the DMA tasks before forwarding them to the physical DMA controller. Thus, programming a DMA task appears to the guest as if there were no virtualization. In the background the hypervisor checks access conditions and takes care of scheduling issues. When the DMA task is done, the hypervisor forwards the interrupt it has received from the DMAC to the guest in charge.

A. Shadow Copies and Scheduling

To prevent guests from interfering during foreign DMAC setups, the hypervisor maintains shadow copies of the DMAC (one for each guest mode). When a guest tries to access a certain register of the physical DMAC, the hypervisor is invoked via the data abort handler and writes the given value into the guest mode's shadow DMAC instead. The physical DMAC will only receive complete and bundled data from the hypervisor. There might exist more guests concurrently interested in DMA than resources are available. As guests may assume that they are possessing the hardware for their own use, they will not actively wait with submitting DMA requests until the DMAC is not longer occupied. Instead, they will post their request assuming that it is processed. Thus, the hypervisor keeps track on which guests have commanded a DMA task and memorizes the parameters as long as the task is not sent to the real DMAC (when occupied). The decision to manage shadow DMACs makes the memorization of parameters simple. A queue is used to schedule the DMA tasks. As we assume a symmetric system, tasks can be assigned to any free channel of the physical DMAC, regardless of whether the physical and the virtual channel numbers are equivalent. Depending on whether a DMA task is enqueued or posted to the physical DMAC, the according virtual channel in the guests's shadow DMAC will be marked as *waiting* or *active*, respectively.

B. Trapping with the Data Abort Handler

After trapping to the privileged ring, the data abort handler of the hypervisor determines from where an access attempt

came and to which address it was directed. To get to know whether it was a reading or writing access and which CPU register was supposed to be involved, the hypervisor decodes the machine instruction. If the instruction refers to the DMAC, the hypervisor will calculate which channel and which register was meant and as long as the (virtual) channel is not marked busy the value to be written will be filled in into the shadow register. In case of an correctly formatted access to the configuration register, the DMA task in question is checked with respect to the access policy.

C. Access Control

The DMA access policy is directly derived from the system's access definitions. Whether or not a guest is allowed to read or write from or to a certain spot in memory is already defined in the MMU coprocessor registers and the access control data structures of the hypervisor. The same rules will apply for copying operations with the use of DMA.

D. Handling DMA Interrupts

The purpose of DMA interrupts is to inform the issuer of a DMA task that the operation is done. In our scenario the hypervisor will receive those interrupts. Besides forwarding them to the corresponding guest mode by calling a special handler provided by the guest, the hypervisor also uses the interrupts to determine when the DMAC is available again. Summarized, the hypervisor safes the execution context of involved guest modes, disables further interrupts and switches to the interrupt handling mode. After finishing its procedure, the just called guest yields back to the hypervisor by a hypercall. Finally, the hypervisor re-enables interrupts, dequeues waiting DMA tasks and returns to the interrupted guest mode.

VI. Evaluation

A. Performance

With the help of the emulator by OVP we compared the performance of the described virtualization solution with DMA support to:

1) the performance of DMA in a non-virtualized (not protected) system and
2) the performance of copying without DMA.

Especially the latter underlines the value of our work. The usage of an emulator allows us to make a very detailed performance analysis based on the granularity of cycle numbers. Table I gives an overview on how many cycles the setup of the DMAC, the programming of a task and the handling of a DMA interrupt take in the standard and the secured virtualized version, respectively. The DMA programming step is detailed analyzed in respect to its substeps: trapping into the hypervisor, obtaining the shadow copy (SC) of the guest mode, filling out the shadow copy, performing the access control check, submitting the task to the real controller and the possible need to enqueue a task. The number of required cycles of different requests can vary slightly, depending on the channel addressed (due to address calculation), the number of channels used, the access policy etc. Therefore, both the minimum and the maximum values of our tests are listed. The upper bounds for setup, programing and interrupt handling, respectively, are compared to the performance values of the non-virtualized reference system. The resulting factors are listed in the fourth column. Our reference processor, the ARM926EJ-S, performs at least 200 million instructions per second (MIPS) so that the required time for the single steps can be approximated (in micro seconds). Even though the factors seem to be quite high at first glance, the security effort is nonetheless reasonable for a solution that does not require any hardware changes to the legacy system at all: a whole DMA cycle (programming a task and handling the interrupt) will not take more than 15 micro seconds. This low overhead in the pre- and postprocessing is negligible compared to the benefits of using DMA (for details see below).

TABLE I
Overhead of securing through virtualization

step	standard	virtualized	×	μsec
setup of DMAC	51	51	1	0.26
programming				
.. trapping		466		
.. obtaining SC		135 - 178		
.. filling out SC		233 - 272		
.. access control		462		
.. submission	44	380 - 458		
.. (enqueuing)		(57)		
= total	44	1676 - 1836	42	9.18
interrupt handling	89	1155	13	5.78
total per task	133	2831 - 2991	23	14.96

Besides comparing non-secured and secured DMA with each other, we also have analyzed when the performance benefits of secure DMA exceed its costs. More specificly, we have measured the CPU cycles required to copy different amounts of data without DMA, that is, only with the CPU running copying instructions in a loop. Figure 1 displays those results ("CPU") and compares them to the constant costs of using DMA ("DMA"). Copying a single word (4 bytes) without DMA requires 21 to 25 cycles on average, depending on the data structure and loop overheads. While for small tasks the use of DMA is not appropriate, from around 128 words (0.5 KB) on it consumes less CPU cycles than the standard way. Note that both variants are secured by the hypervisor. Finally, the graph labeled "DMA/CPU" shows the ratio of the costs when copying with DMA and CPU only, respectively. It is clearly visible that the use of DMA quickly becomes strongly preferable, even without loss of security.

B. Security Analysis

Referring back to the security requirements defined in Sect. III-C, we now go through them step by step (original numbering kept) and reason why our solution fulfills them.

1) The memory region in which the DMAC is situated is configured to deny any access from the unprivileged processor mode in which all guest code is executed. Every access attempt invokes the data abort handler defined by the hypervisor.

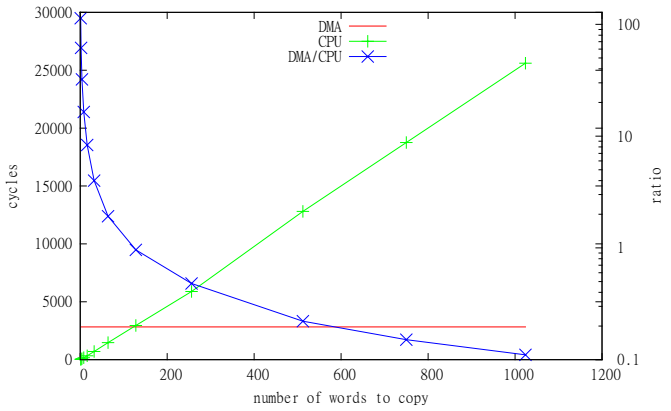

Fig. 1. Comparison between DMA and CPU copying of words in a secured environment

2) The hypervisor is the only entity which is able to program the DMAC. Before filling out its registers, the DMA task in question is checked according to the same criteria which apply to writing and reading processes without DMA support. Should this test fail, the task will neither be submitted nor enqueued. Once a task is actually submitted or enqueued, it cannot be modified anymore.
3) All programming steps are performed on a guest specific shadow DMAC first. That is, no guest can interrupt another one and modify specifications such as source and destination address in the name of the interrupted guest. This protection ensures that interferences during the programming of the DMAC are not possible. Only the very last step, the actual commitment, causes the physical DMAC to be filled out. This is done by the hypervisor on the basis of the shadow DMAC of the current guest. With other words, no data from other guests can influence this process. Interrupts are disabled during this last step.
4) The queuing operations do not modify the actual content of DMA tasks. Only the physical channel chosen can differ from the virtual channel number, which does neither affect which data is copied nor to which address. Furthermore, the hypervisor keeps track from which guest a certain DMA task came so that interrupts are forwarded to the actual issuer of a DMA request.
5) The memory region in which the hypervisor is situated is configured to deny any access from the unprivileged processor mode, in which all guest code is executed. Therefore, the protection of the hypervisor itself follows from observation 2.
6) The number of bytes to copy and the source and destination address of a DMA request communicated by the guest are saved without modifications in the shadow structure of the guest mode. As each guest mode has an own shadow copy, no interference between guests can occur. The request procedure is completed by a write attempt to the configuration register. This will invoke the access policy check. In case of acceptance, the task is submitted to the DMAC or, if this is occupied, will be enqueued. The queue can not overflow as the number of DMA tasks marked waiting is limited by the number of guest modes times the number of DMA channels. This follows from the fact that each channel can only be programmed once at a time per guest mode.
7) First, the characteristics of a queue ensures that each task enqueued will be processed at some time. Invalid requests will not even be enqueued. Tasks are finite and after their completion the DMAC will fire an interrupt, which will reach the hypervisor. The latter will delete the task from the DMAC and its internal structures. If an interrupt occurs while another one is processed, the hypervisor will notice the second task completion as well as it (re-)checks the states of all DMAC channels immediately before leaving the interrupt service routine again. However, denial of service attacks can be a threat to the complete handling of DMA interrupts. How they can be prevented or at least limited is described below.

The possibility that a guest does not yield back to the hypervisor is always given, not only in the context of DMA support. A good way to get control back in any case is the use of a timer tick counter or watchdog interrupt, which will call a hypervisor function after a predefined amount of time. Especially for the DMA interrupt handling presented here there are additional means by extending the configuration of the hypervisor. For example, the hypervisor can be changed to disallow or postpone the interrupting of guest modes which are seen as especially important and protect worthy. That way, a malicious guest mode cannot use a DMA task to get execution time (via its DMA interrupt handler) during a sensitive operation of another guest mode. Vice versa, it is conceivable to grant only trusted guests the possibility of own DMA interrupt handlers, at least in critical situations. But not only the DMA interrupts need to be considered. Each attempt to program the DMAC, no matter if successful or not, causes a delay in the system. To prevent malicious guests to use this for slowing the system down, the hypervisor can easily be modified to restrict accesses to a certain number or frequency per guest mode.

VII. Formal Verification of a Simplified DMA Model

One of the benefits of using virtualization for security is the relatively small size of a special purpose hypervisor compared to a complex modern operating system. This reduces the trusted computing base distinctly and allows formal verification of the system's overall security properties. We used the *Coq* theorem prover [19] to show memory isolation in the context of DMA on a highly simplified model.

Assuming there are exactly two guests, either of them potentially malicious. Guest 2 is the "object of comparison": it is shown that the memory region and data structures assigned to guest 2 are not influenced by DMA operations of guest 1. It follows that both the integrity of guest 2 and the confidentiality

of guest 1 hold.[1] We model the machine state as a record of a simplified memory, the DMAC and its two shadow copies, flags indicating which guest system has DMA transfers active, resp. waiting, and finally an interrupt flag indicating the completion of a DMA transfer. We assume that a guest system can read the memory addresses belonging to it at any time. In contrast, the MMU prevents accesses (not using DMA) to the other guest's memory. An execution consists of a sequence of state transformations. Three state transformations are represented in the model, namely the activities of the hypervisor initialized whenever a guest attempts to access the DMAC, the operations performed by the DMAC and the functionality of the interrupt handler. As those transformations usually occur together, we summarize such a chain to an execution block, denoted by F. Depending whether it was evoked by guest 1 or 2, it is referred to by F_1 or F_2, respectively. Following those conventions the execution of the system can be seen as a sequence of blocks, as for example $F_1 - F_2 - F_1 - F_2 - F_2 - F_2 - F_1 - \ldots$. We show that, in this model, under no circumstances the content of the memory region for guest 2 will depend on the inputs made by guest 1. With other words no information can flow from guest 1 to guest 2. This is done by proving that for any execution sequence, neither the memory content belonging to guest 2 nor the content of guest 2's shadow DMAC registers change when eliminating blocks of type F_1. The proof uses an important state invariant, reflecting that the state is *correctly and soundly configured*, defined as follows:

- The DMAC is set up according to the access policy.
- If some shadow DMAC is marked enabled, its source and destination registers follow the access policy.
- No guest is active and waiting at the same time.
- The interrupt flag is only set if the DMAC is enabled.
- The physical DMAC is enabled if and only if at least one of the shadow DMAC is.
- A shadow DMAC is marked enabled if and only if the corresponding guest is marked as *active* or *waiting*.
- If some guest is marked as *waiting* then the other guest is marked as *active*.

It is shown that all hardware and hypervisor operations maintain this invariant, that is, when starting in a correct and soundly configured state, the execution will result in another correct and soundly configured state. Finally the main theorem is formulated as follows: Starting in a correct and sound configured initial state and building up two execution sequences of which one represents any actual execution and the other is the related sequence ignoring all operation/execution blocks caused by guest 1, then at every step both sequences will be equivalent with respect to guest 2. We were successful in proving the theorem in Coq using 2200 lines of proof, out of which 300 describe the actual model. Even though this initial verification effort demonstrates that the DMA approach of the paper is based on sound fundamentals, it still relies on strong assumptions and a very simplified model. Work on refining the approach is in progress.

[1]Confidentiality of guest 1 and integrity of guest 2 hold by symmetry.

VIII. Conclusions

We have demonstrated that DMA virtualization only based on software and standard hardware can provide high security guarantees. To strengthen this even more we also have applied formal verification. Besides isolation, availability is warranted. Moreover, our approach has a low performance overhead, on the order of a few microseconds in pre- and postprocessing of DMA transfers. We intend to compare our performance results to systems with ARMv7 and/or System MMU. Additional features such as broader platform support or secure booting are also left for future work. As for verification, we plan to analyze the whole hypervisor on binary level.

References

[1] C. Gehrmann, H. Douglas, and D. K. Nilsson, "Are there good reasons for protecting mobile phones with hypervisors?" in *IEEE Consumer Communications and Networking Conference*, January 2011.

[2] ARM, "Cortex-A15 processor," http://www.arm.com/products/processors/cortex-a/cortex-a15.php.

[3] R. Mijat and A. Nightingale, "Virtualization is coming to a platform near you," *ARM Whitepaper*, 2011, http://www.arm.com/files/pdf/System-MMU-Whitepaper-v8.0.pdf.

[4] Advanced Micro Devices, Inc., "AMD I/O Virtualization Technology (IOMMU) Specification," 2009, http://www.mimuw.edu.pl/~vincent/lecture6/sources/amd-pacifica-specification.pdf.

[5] J. Brakensiek, A. Dröge, M. Botteck, H. Härtig, and A. Lackorzynski, "Virtualization as an enabler for security in mobile devices," in *Proceedings of the 1st workshop on Isolation and integration in embedded systems*, ser. IIES '08. ACM, 2008, pp. 17–22.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, ser. Operating Systems Review, vol. 37, 5. New York: ACM Press, Oct. 19–22 2003, pp. 164–177.

[7] J.-Y. Hwang, S.-B. Suh, S.-K. Heo, C.-J. Park, J.-M. Ryu, S.-Y. Park, and C.-R. Kim, "Xen on ARM: System Virtualization using Xen Hypervisor for ARM-based Secure Mobile Phones," in *5th IEEE Consumer Communications and Networking Conference (CCNC 2008)*, Las Vegas, NV, USA, Jan. 2008.

[8] F. Armand and M. Gien, "A Practical Look at Micro-Kernels and Virtual Machine Monitors," in *Proceedings of the 6th Consumer Communications and Networking Conference (IEEE CCNC '09)*, Las Vegas, NV, USA, Jan. 2009.

[9] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes," in *Proceedings of the 21st Symposium on Operating System Principles(SOSP 2007)*, Stevenson, Washington, USA, Oct. 14–17 2007, pp. 335–350.

[10] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "TrustVisor: Efficient TCB reduction and attestation," in *Proceedings of the IEEE Symposium on Security and Privacy*, May 2010.

[11] H. Härtig, J. Löser, F. Mehnert, L. Reuther, M. Pohlack, and A. Warg, "An I/O Architecture for Mikrokernel-Based Operating Systems," Dresden University of Technology, Dresden, Germany, Tech. Rep. TUD-FI03-08, 2003, http://os.inf.tu-dresden.de/papers_ps/tr-ioarch-2003.pdf.

[12] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," in *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*. Big Sky, MT, USA: ACM, October 2009, pp. 207–220.

[13] H. Tews, "Micro hypervisor verification: Possible approaches and relevant properties," 2007.

[14] U. Steinberg and B. Kauer, "NOVA: A Microhypervisor-Based Secure Virtualization Architecture," in *Proceedings of EuroSys*, 2010.

[15] W. Arbaugh, D. Farber, and J. Smith, "A secure and reliable bootstrap architecture," *IEEE Symposium on Security and Privacy*, p. 0065, 1997.

[16] F. L. Sang, ric Lacombe, V. Nicomette, and Y. Deswarte, "Exploiting an I/OMMU vulnerability," in *5th International Conference on Malicious and Unwanted Software (MALWARE)*, 2010, pp. 7–14.
[17] "Open virtual platforms," http://www.ovpworld.org.
[18] *PrimeCell DMAController (PL080) Technical Reference Manual*, rev r1p3 ed., ARM, 2005, http://infocenter.arm.com/help/topic/com.arm.doc.ddi0196g/index.html.
[19] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development : Coq'Art: The Calculus of Inductive Constructions*. Berlin: Springer-Verlag, 2004.