EPiC Series in Computing
Volume 49, 2017, Pages 1–16
PROOFS 2017. 6th International Workshop on Security Proofs for Embedded Systems

Compositional Verification of Security Properties for Embedded Execution Platforms

Christoph Baumann[1], Oliver Schwarz[2], and Mads Dam[1]

[1] KTH Royal Institute of Technology, Stockholm, Sweden – {cbaumann,mfd}@kth.se
[2] RISE SICS, Kista, Sweden – oliver.schwarz@ri.se

Abstract

The security of embedded systems can be dramatically improved through the use of formally verified isolation mechanisms such as separation kernels, hypervisors, or microkernels. For trustworthiness, particularly for system level behavior, the verifications need precise models of the underlying hardware. Such models are hard to attain, highly complex, and proofs of their security properties may not easily apply to similar but different platforms. This may render verification economically infeasible. To address these issues, we propose a compositional top-down approach to embedded system specification and verification, where the system-on-chip is modeled as a network of distributed automata communicating via paired synchronous message passing. Using abstract specifications for each component allows to delay the development of detailed models for cores, devices, etc., while still being able to verify high level security properties like integrity and confidentiality, and soundly refine the result for different instantiations of the abstract components at a later stage. As a case study, we apply this methodology to the verification of information flow security for an industry scale security-oriented hypervisor on the ARMv8-A platform. The hypervisor statically assigns (multiple) cores to each guest system and implements a rudimentary, but usable, inter guest communication discipline. We have completed a pen-and-paper security proof for the hypervisor down to state transition level and report on a partially completed verification of guest mode security in the HOL4 theorem prover.

1 Introduction

The rise of embedded systems and the internet of things has been accompanied by a surge of cyber-attacks against them. A possible solution to this security problem is to design provably secure systems on top of formally verified separation kernels and hypervisors that provide isolation guarantees through virtualization and help to reduce the trusted computing base.

Reflecting this trend towards increased use of virtualization, hardware vendors have started to provide hardware virtualization support also for embedded system processors and system-on-chips (SoCs). Tasks previously done in software now rely on this hardware and its correct configuration. At the same time, devices and low-level hardware components like caches [18] and direct memory access (DMA) controllers have also been identified as potential attack surfaces. For instance, an adversary controlling a DMA device (e.g. via a driver [36]) might be able to circumvent the kernel's memory isolation and install stealthy keyloggers [33]. Input/output

memory management units (IOMMUs) allow the kernel to constrain the address ranges accessible by devices. However, IOMMUs are not always free from vulnerabilities either [28] and proper configuration is not entirely trivial. Whatever protection system designers choose, it is crucial to include hardware attack surfaces, protection units, and the configuration of both into the reasoning, when system software is formally verified. However, more complete models of the underlying hardware are by nature hard to attain and complex, leading to costly proofs of their security properties. Moreover, if a model is monolithic and specific to a given platform, proofs may not easily apply to similar but different platforms and the verification easily becomes economically infeasible.

To overcome these issues, we propose a compositional top-down approach and model a SoC as a network of distributed automata that communicate using paired synchronous message passing. Abstract specifications for each component allow to delay the development of detailed models for cores, devices, etc., while still being able to verify high level security properties like integrity and confidentiality. Subsequently, abstract components can be instantiated with more refined models and overall security is preserved by discharging local verification conditions identified in the top-level proof. Decomposition provides the further advantage that guarantees on constant parts can be reused when other parts change. Such reusability and adaptability is especially important for embedded systems, since standard chipsets are rare and verification needs to be performed for many different custom SoC designs.

As a case study, we apply the methodology to the verification of information flow security for an industry scale security-oriented hypervisor on ARMv8 [7]. The hypervisor, developed in the open source HASPOC project [19], provides full virtualization and supports several versions of Linux (Debian, Ubuntu) and Android running on the HiKey 96-boards platform based on the 8-core HiSilicon Kirin 620 Cortex-A53 SoC. The hypervisor statically assigns (multiple) cores to each guest system and implements a rudimentary, but usable, inter guest communication discipline. The verification focuses on the behavior of the underlying SoC hardware during guest execution. In particular, we consider memory, peripherals, (IO)MMUs, cores – including their user mode capabilities – as well as interrupt controllers. Models and proofs have been formalized in the HOL4 theorem prover with promising preliminary results.

2 Related Work

The merits of compositional reasoning in the design and validation of embedded systems have been acknowledged for a long time, c.f. [16, 31, 4, 37, 21]. A compositional approach based on component abstraction and rely-guarantee reasoning is also at the core of the contract-based design and verification paradigm [27, 10]. In this work, we apply the same underlying techniques to the formal verification of security properties for the low-level execution platform, that are established if the hardware is configured properly by a trusted or verified piece of software, e.g., a hypervisor.

The first verification exercises of system software date back several decades [13]. The research discipline has gained increased traction in recent years through prominent projects such as seL4 [22] and Verisoft (XT) [3, 25]. Since isolation is both enabled (e.g. by MMUs) and threatened (e.g. by DMA) by hardware, it is crucial to include underlying hardware into the reasoning. This gains even more importance with virtualization support that shifts tasks traditionally managed by software to hardware units such as 2-stage MMUs. Given the central role of memory management, recent work modelled the effects of several kinds of MMUs, their proper configuration, caches, TLBs, and their interplay with system software [5, 8, 24, 34, 6]. The formalization of peripherals has been done both from a functional and from a security

perspective. For security, the main concern is the preservation of memory isolation in the presence of DMA devices [9]. Kernel verification has been studied both for settings with IOMMUs [35, 20, 17] and for peripherals configured to comply with constrained access policies [29]. Finally, kernel code is not the only code executing on the system's processors. Instruction sets might grant to low-privileged code access to sensitive resources that kernel designers are not always aware of. While it is possible to mitigate such threats, it is important to be aware of them both in the design and verification of kernels. ARM started to create machine-readable ISA specifications [26] and demonstrated how to exploit them to check noninterference properties of the ARMv8-M security extensions. Similarly, the information flow behavior of ARMv7 user mode execution has been analyzed in [30]. While the above examples focus on single system parts, we are interested in a holistic view and system-wide isolation guarantees, where existing results potentially can be reused as building blocks.

3 System Model

Our system model is characterized by a tuple $\langle n, \mathbb{M}, \mathcal{K}, \mathcal{T} \rangle$, where $n > 0$ is the number of components, $\mathbb{M}$ is the type of their messages, $\mathcal{K} : \mathbb{N}_n \to \mathbb{K}$ contains all the component specifications (with $\mathbb{N}_n = \{1, \ldots, n\}$), and $\mathcal{T} \subseteq \mathbb{T}$ represents the possible global transitions of the system by a set of synchronization vectors. Let $\mathbb{B} = \{0, 1\}$. A component specification $k \in \mathbb{K}$ is a record with the following components:

- Σ – the set of component states,
- $\Sigma_0 \subset \Sigma$ – a set of possible initial states for the component,
- $\mathrm{snd}, \mathrm{rcv} : \Sigma \times \mathbb{M} \times \Sigma \to \mathbb{B}$ – transition relations for sending and receiving messages $m \in \mathbb{M}$,
- $\tau : \Sigma \times \Sigma \to \mathbb{B}$ – a transition relation for internal steps.

Note that we use transition relations (defined as boolean predicates) instead of functions to a next state, in order to not rule out non-deterministic component models. Communication between the components is governed by synchronization vectors of type $t \in \mathbb{T}$, defined as follows:

$$t \quad ::= \quad \textsc{Msg}\ i\, j\, m \mid \textsc{Tau}\ i \mid \textsc{ExtI}\ i\, m \mid \textsc{ExtO}\ i\, m\,.$$

A vector $\textsc{Msg}\ i\, j\, m$ denotes that a message m may be sent from component i to j, assuming $i \neq j$. Vector $\textsc{Tau}\ i$ represents internal actions of component i. Similarly, vectors $\textsc{ExtI}\ i\, m$ and $\textsc{ExtO}\ i\, m$ represent external I/O actions of component i with associated messages m.

A (global) system state $s \in \mathbb{S}$ is now a mapping from component indices $i \in \mathbb{N}_n$ to component states $s(i) \in \mathcal{K}(i).\Sigma$. For all transitions $t \in \mathcal{T}$ a corresponding system transition from state s to s' is denoted by $t \vdash s \to s'$ and defined in Fig. 1. There the global system transitions are mapped to local component actions in the obvious way. In particular, a message passing transition $\textsc{Msg}$ is only enabled globally, if both the sending and receiving actions are enabled in the corresponding components. Computations $s \to^n s'$ of the system model for n steps are defined by demanding that there exists a sequence of n transitions $t_1, \ldots, t_n \in \mathcal{T}$ that transform s into s' by repeated application of the transition relation.

3.1 Instantiation and Soundness

When instantiating the system model outlined above, a "natural" decomposition of a SoC can be conducted as follows. As message channels we identify memory buses and interrupt signals

$$\frac{i,j \in \mathbb{N}_n \quad i \neq j \quad \mathcal{K}(i).\mathrm{snd}(s(i),m,s'_i) \quad \mathcal{K}(j).\mathrm{rcv}(s(j),m,s'_j)}{\textsc{Msg}\ i\,j\,m \vdash s \rightarrow s[i \mapsto s'_i; j \mapsto s'_j]}\ \textsc{Msg} \qquad \frac{i \in \mathbb{N}_n \quad \mathcal{K}(i).\tau(s(i),s'_i)}{\textsc{Tau}\ i \vdash s \rightarrow s[i \mapsto s'_i]}\ \textsc{Tau}$$

$$\frac{i \in \mathbb{N}_n \quad \mathcal{K}(i).\mathrm{rcv}(s(i),m,s'_i)}{\textsc{ExtI}\ i\,m \vdash s \rightarrow s[i \mapsto s'_i]}\ \textsc{ExtI} \qquad \frac{i \in \mathbb{N}_n \quad \mathcal{K}(i).\mathrm{snd}(s(i),m,s'_i)}{\textsc{ExtO}\ i\,m \vdash s \rightarrow s[i \mapsto s'_i]}\ \textsc{ExtO}$$

Figure 1: Overall system semantics. Component i in state s is updated to s'_i using notation $s[i \mapsto s'_i]$ for the resulting system state.

and we model each bus master, e.g., cores and devices, as separate components. Similarly, the memory system, including caches and RAM, is modeled as a component of its own. External inputs and outputs target the devices only and are specific for each device, modeling for instance network packages, user input, sensor data, or actuator control signals.

Note that communication through memory buses and interrupt signals is in nature asynchronous, i.e., a sender does not know when a message will be received or a potential reply be returned. In order to synchronize communication between components, these channels can either be modeled as separate buffering components that communicate synchronously with the sender and receiver components, or be merged with either the sender or the receiver, e.g., the memory bus with the memory system or a buffered interrupt signal with the sending device. Still, components in such a system may be quite complex, prompting further decomposition, e.g., to separate MMU functionality from a core, or caches from memory.

Concerning communication, receiving transitions in such a system can occur whenever a component is ready to receive data from a memory bus or an interrupt. Conversely, sending transitions occur whenever a component is ready to send data or an interrupt. If a component originally both receives and sends information simultaneously, the behavior must be modeled by two separate transitions, where the receiving transition is executed first and then blocks until the sending transition is completed. As defined above, the synchronized send and receive transition represent atomic actions in our decomposed SoC model. When defining the component transitions, care must be taken wrt. atomicity, i.e., not to rule out certain interleavings of actions that lead to externally observable behavior. Therefore, internal transitions are important to control the granularity of component transitions and expose certain intermediate component states that are observable by other components or the environment through communication.

In general, the decomposed model and its interleaving semantics are sound if its externally observable behavior is identical to that of a monolithical or true-concurrent model. As computer systems are discrete, and external communication is generally asynchronous, the exact timing of signals and component transitions can usually be neglected so that the interleaving model captures all system behaviors. If timing properties are important, time can be added as a variable to the model and restrict the possible interleavings [2]. Moreover, if completeness of the model is desired, guards on component transitions and restrictions on the global schedule may be necessary to rule out behaviors in the interleaved semantics that are not possible in the real system.

3.2 Abstraction

The compositional approach facilitates *abstraction*. While detailed component models may be complex, it often is not necessary to expose all of this complexity to a top-level proof. Having decomposed the system into communicating units with clear interfaces opens the door to underspecifying internals and focusing on interactive component behavior. Formally, we

introduce an abstract component specification $\hat{\mathcal{K}}$ and surjective abstraction functions $\text{abs}_i : \mathcal{K}(i).\Sigma \to \hat{\mathcal{K}}(i).\Sigma$ for all $i \in \mathbb{N}_n$ such that:

$$\begin{aligned} \mathcal{K}(i).\text{rcv}(\sigma, m, \sigma') &\Leftrightarrow \hat{\mathcal{K}}(i).\text{rcv}(\text{abs}_i(\sigma), m, \text{abs}_i(\sigma')) \\ \mathcal{K}(i).\text{snd}(\sigma, m, \sigma') &\Leftrightarrow \hat{\mathcal{K}}(i).\text{snd}(\text{abs}_i(\sigma), m, \text{abs}_i(\sigma')) \\ \mathcal{K}(i).\tau(\sigma, \sigma') &\Leftrightarrow \hat{\mathcal{K}}(i).\tau(\text{abs}_i(\sigma), \text{abs}_i(\sigma')) \,. \end{aligned}$$

These conditions ensure that all properties proven about the abstract model $\hat{\mathcal{K}}$ also hold on the concrete one $\mathcal{K}$ (for parts of the state covered by the abstraction function) [1]. Note that equivalence is only needed in order to transfer trace hyperproperties, such as confidentiality [11]. For safety properties, the implication from detailed to abstract model is sufficient.

This abstraction technique allows to start verification with an abstract model and only later develop a more detailed model that refines the abstract one, reducing the initial modeling effort. In practice, i.e., our case study, we have experienced that specifications for components can be made very abstract using uninterpreted and underspecified types and functions, without losing expressiveness wrt. the verification of platform security properties. We continue detailing our modeling and verification methodology using the case study as an example.

4 Case Study: ARMv8 Platform Security

Below we present our case study of the compositional verification approach. We verify security properties of an ARMv8-based execution platform as established through configuration by a minimal bare-metal hypervisor which hosts a number of untrusted guests. The hypervisor statically partitions the system resources, i.e., cores, memory, devices, and interrupts between the guests, i.e., there is no resource sharing, except through a predefined set of communication channels. The top goal of the verification effort is to show information flow security for the hosted guest systems, i.e., that information can only be exchanged between guest partitions through allowed channels. To this end, it is also necessary to prove integrity of the hypervisor, as any successful attack on it may break the isolation guarantees imposed on the system.

The security property is formulated in the form of a bisimulation theorem between the decomposed ARMv8 *platform model* and an idealized specification where guest systems are running on dedicated SoCs with explicit communication channels between them [12]. Both models and the hypervisor design are completely formalized in the HOL4 computer aided verification system. While the verification effort is ongoing, for the part of the bisimulation concerning guest execution, i.e., the steps not virtualized by hypervisor handlers, we already verified the most challenging cases with reasonable effort.

4.1 ARMv8 Platform Model

As the basis for our modeling work we extended the user-level ARMv8 CPU model by Fox [15] with system-level functionality, i.e., the register state and instructions for the hypervisor and TrustZone execution modes, as well as virtualization extensions in form of a two-stage MMU. However, for a complete SoC model, detailed models of the System MMU (SMMU, aka IOMMU), the Generic Interrupt Controller (GIC), the memory subsystem, and all of the devices were missing. Nevertheless, the CPU model was already designed with decomposition in mind, consisting of separate automata for the core instruction execution, separate first and second stage MMUs, and a flat main memory, communicating via message passing.

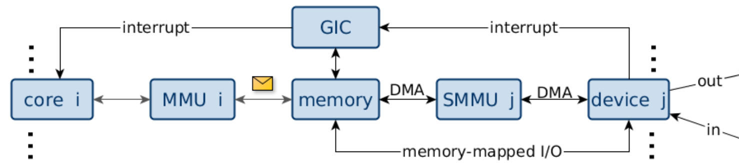

Figure 2: ARMv8 platform model

In such a situation the combined approach of decomposition and abstraction shows its strengths. As a first step we decompose the SoC into the following components: (1) a parameterized number of ARMv8 cores including their first stage MMUs, (2) the corresponding second-stage MMU for each core, (3) a shared main memory component, (4) a parametrized number of arbitrary devices, (5) a corresponding SMMU component for each device, and (6) the GIC, that is treated as a special kind of device. There is another special device, the power controller used for starting and stopping cores, but we omit its description here for brevity.

The first stage MMUs are merged with the cores, because in a hypervisor scenario they are completely controlled by the untrusted guests, therefore their interactions with the cores are irrelevant to the overall system security and we chose to simplify the model. In what follows, we will refer to second stage MMUs simply by "MMU". Moreover, having one SMMU per device may seem like a strong assumption. Nevertheless, modern SMMUs usually manage different session IDs for different devices, hence they can be modeled as private SMMUs for each device.[1]

The possible communication channels between the components are shown in Fig. 2 and we distinguish (1) memory requests and replies, (2) virtual and physical interrupts, and (3) external input and output signals for the devices, as messages of our system model. Memory requests can be reads, writes, or page table walks of some (S)MMU. Memory replies contain either a result for a matching request or a fault, e.g., due to failed access permission checks in the MMU. In this case study we do not cover special memory instructions like barriers or cache flushes.

Reflecting a modeling decision in our initial CPU model, the MMU automaton handles all communication between core and memory. Similarly, devices access the memory via their SMMU. We decided not to model the memory bus explicitly. It is integrated into the memory component, which thus needs to distinguish regular memory accesses from memory-mapped I/O (MMIO) accesses by the cores and forward the latter to the right device (not involving any SMMU). Devices may send interrupts to the GIC, from where they are forwarded to the cores according to the GIC configuration. Cores can also request software-generated interrupts (SGIs) to other cores through MMIO accesses to the GIC distributor module and the hypervisor can configure virtual interrupts for the guests through accesses to the corresponding GIC interface.

The component configurations in our system model are instantiated with abstract specifications instead of detailed models, adapting the level of detail to the requirements of the top level security proof. Specifically, the component state is kept as abstract as possible. To keep track of sent and received memory requests, as well as received memory replies, all abstract component states are equipped with corresponding history variables. Moreover, instead of defining the component transition relations explicitly, each transition is described by a collection of specification functions that relate the pre and post states of the transition for different cases. Almost

[1] Note that this model requires a correctness proof, showing that one SMMU virtualizes several SMMUs with different session IDs. We assume it here for the sake of a simpler model.

all components can perform internal transitions, which are mostly underspecified except for history variable updates. Further details on the components are described below.

Core and first stage MMU We model explicitly the program counter and processor status register (containing among others the execution level). The remaining register state is divided into an uninterpreted guest register state and a hypervisor register state. The guest register state contains all registers accessible in execution levels EL0 and EL1, e.g., general purpose registers, as well as system and status registers controlling the first stage MMU. The hypervisor register state contains control registers accessible in EL2 and EL3, we only model those relevant for the hypervisor design, e.g., HCR_EL2 and SCR_EL3, controlling traps from guest mode.

The possible transitions of the core are: (1) sending/receiving a memory request and (2) receiving a virtual or physical interrupt. All transitions that do not change the execution mode are largely underspecified, e.g, for sending memory requests we assume that all information in guest registers besides the execution mode may change arbitrarily. We only demand that the history variable recording sent requests is updated correctly and that hypervisor registers are unchanged. Memory replies may be received from the MMU if there is a matching outstanding request and they are modeled similarly to send transitions unless a fault is received. Then, and when receiving interrupts, an exception occurs. If the mode changes to EL2 or higher, we model the behavior precisely, in order to identify the responsible hypervisor handler.

Second stage MMU The detailed model of the ARMv8 memory management units is quite complex, exhibiting a large number of different address translation schemes and corner cases. However, if configured statically by the hypervisor this complexity can be handled by representing the explicit MMU configuration as an abstract translation scheme parametrized for each guest and by keeping track of the translation status for pending memory requests. In particular, we keep an abstract MMU state that records for every possible memory request if a corresponding translation is idle, i.e., not requested, still translating, or in its final stage where the translated request is forwarded to memory. While a request is being translated, we consider it non-deterministic when it will reach its final state or result in a fault. We only require a progress condition that it will do so after a finite amount of MMU steps. Possible actions of an MMU are (1) receiving a memory request from its core, (2) sending a translation table lookup or a final memory request to memory, (3) receiving a corresponding reply from memory, and (4) replying to the core for a pending request: either with a fault or the result from memory. While the hypervisor is running on a core, we model its MMU as being turned off. Then requests from the core are just forwarded to memory without translation.

Memory In this case study we use a simple coherent shared memory model without caches or weak memory ordering. This is reflected in our abstract memory specification, which consists of a single page-addressable map of physical memory contents along with the message history variables. The transitions of memory consist of (1) receiving a request from an MMU or SMMU (in case of device DMA access), (2) forwarding a core's MMIO access to a device, (3) receiving a device's reply to an MMIO access, and (4) sending a reply for an earlier memory request. In the latter case for physical memory access the usual memory semantics apply, MMIO replies from devices are simply forwarded.

Soundness of the memory model presupposes that neither the hypervisor nor the guests break memory coherency, and that the hypervisor code itself is correctly synchronized. The former assumption can in fact be broken by phenomena such as mismatched cache attributes that are known to produce memory incoherency and to potentially break guest-guest as well

as guest-hypervisor isolation [18]. The latter assumption (that the hypervisor code itself is correctly synchronized) is delicate to validate in general but goes outside the analysis presented in this paper. Weakly consistent memory behavior as seen in ARMv8 processors does not seem to be a security concern. It mainly seems to complicate the verification of hypervisor synchronization primitives. For guest memories, the memory semantics can easily be replaced with more realistic models along the lines of previous work [14].

Devices Since all DMA accesses are protected by the SMMUs and the guests have full control over the devices without the hypervisor ever touching them, we can leave the device states completely uninterpreted. Device transitions consist of (1) receiving and replying to MMIO accesses from memory on behalf of a core, (2) sending DMA requests to an SMMU and receiving DMA replies, (3) sending or receiving external signals associated with that device, and (4) sending an interrupt associated with that device to the GIC. Our synchronous message passing approach fits well with edge-triggered interrupt signaling. However, we can also model level-triggered interrupts with the same mechanism, it all depends on the GIC's interpretation of the edges.

SMMU On a detailed level the SMMUs on ARM SoCs differ from the regular MMUs of the cores. However, the translation mechanism is similar and on an abstract level both MMUs and SMMUs implement the same kind of functionality. Hence, we use the same abstract specification for SMMUs as for MMUs, albeit parametrized with different translation schemes[2], depending on the guest a device belongs to. It is a particular strength of our approach to allow the reuse of abstract models for components with similar functionality, but different implementation.

GIC The generic interrupt controller used on our ARM platform (GICv2) consists of four different register states: (1) the interrupt distributor shared by all cores, and for each core (2) a physical interrupt interface, (3) a virtual interrupt control interface, and (4) a virtual interrupt interface. In the abstract specification we largely leave the registers underspecified.

For modeling the side effects of MMIO accesses to the GIC and interrupt reception, we introduce uninterpreted functions that map the distributor register contents to the physical interrupt state for the whole system and the control interface registers to the virtual interrupt state per core. Then for the possible GIC actions of (1) receiving an interrupt from a device, (2) signaling a physical or virtual interrupt to a core, (3) receiving MMIO accesses from the cores to one of the register states, and (4) replying to an MMIO access, the side effects of such transitions are expressed rather on the interrupt state than on the register state which can change either non-deterministically or is unaffected by a given transition. Only for the registers explicitly touched by the hypervisor, we model the effect on the registers explicitly. For instance, as the virtual interrupt interface is only used by the guest, we leave the effect on corresponding registers unspecified and overapproximate the side effects of MMIO accesses on the interrupt state. In particular, accesses to the virtual interrupt interface only ever affect interrupts pending or active on that interface and inactive interrupts stay inactive.

For interrupt signaling, the GIC transition is synchronized with a corresponding receiving transition at the targeted core. Thus the signaling is modeled to occur in sync with the core taking the exception for the asynchronous interrupt. We only distinguish physical from virtual interrupts, which can only be received in guest mode, and only model the IRQ interrupt signal.

[2]For simplicity of the model, we forbid DMA accesses to other devices and the GIC in this work.

4.2 Hypervisor Model

The main functionality of the hypervisor is to bring up the platform into a state where different guest systems can run in statically allocated partitions, each owning a number of cores, devices and their interrupts, as well as a region of memory exclusively. Inter-guest communication (IGC) is allowed only via predefined shared memory channels and associated inter-processor notification interrupts.

For this work we are mainly interested in the platform invariant $I : \mathbb{S} \to \mathbb{B}$ that constrains all component configurations of the SoC during system execution to enable the desired information flow policy and guarantee hypervisor integrity. To name a few of the most important properties:

- All translated requests sent by an (S)MMU are constrained by the translation scheme of the corresponding guest, i.e., translated requests from cores and devices may access only the associated guest's physical memory, including outgoing IGC channels and memory mapped I/O regions of owned devices, as well as each owned core's GIC virtual interrupt interface.
- Likewise, all pending requests in memory from cores running in guest mode or devices address the associated guest's region of memory. Forwarded MMIO requests are sent by cores with the same owner as the targeted device and addresses match the device's I/O region. Requests to GIC registers other than the virtual interrupt interface are sent by cores running in hypervisor mode. For replies traversing the platform, similar restrictions hold.
- Page tables are stored in hypervisor memory, that is disjoint from guest memory. They are fixed and implement a secure translation scheme for guests, guaranteeing memory isolation.
- The GIC is configured in such a way that physical device interrupts can only be forwarded to cores of the same associated guest. SGIs are only pending or active between cores of the same guest, unless they have a special ID reserved for IGC notifications. Similar conditions are imposed on the virtual interrupts that are pending or active at each core's interface.
- Well-formedness invariants on messages and the abstract state of all SoC components.
- Additional invariants capture intermediate states of the hypervisor computation, in particular they restrict the value of hypervisor system registers and its internal data structures.

The invariant has to be preserved by all guest and hypervisor steps for all components of the SoC. For initial system states $s_0 \in \mathbb{S}_0$ with $s_0(i) \in \mathcal{K}(i).\Sigma_0$ for $i \in \mathbb{N}_n$ we have to prove:

Theorem 1. *All computations $s_0 \to^n s'$ starting in an initial system state $s_0 \in \mathbb{S}_0$ preserve the platform invariant, i.e., $I(s')$ holds.*

The theorem is proven by induction on n. In particular the invariant needs to be defined in such a way that $I(s_0)$ holds for all such s_0, requiring certain invariant parts only after they have been established by different phases of the hypervisor initialization process. In the induction step we distinguish all possible component transitions and use their abstract specifications. Decomposition allows local reasoning here as at most two components are changed in one step.

For example, in the proof of memory isolation we can focus on the interplay of (S)MMU and memory, arguing that (1) the MMU is configured correctly to only send translation requests to the area where the hypervisor stores the secure second level page tables, (2) memory returns the correct values of data stored in it, i.e., entries from the secure page table, and (3) if the MMU translates a guest's memory request successfully, only using entries from the secure page table for that guest, the translated memory request addresses the guest's memory region.

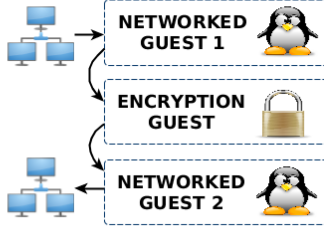

Figure 3: Use case: with correctness of the crypto service, traffic between the networks is guaranteed to be encrypted, even with other guests compromised.

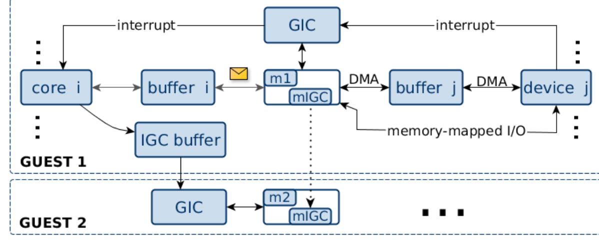

Figure 4: Ideal model: Guest 1 with its share of cores, devices, and memory regions – connected to guest 2 via an inter-guest communication (IGC) interrupt and duplicated but synchronized IGC memory.

In addition to platform intialization, the hypervisor contains handlers for providing virtualized functionality to the guest. In particular, it (1) virtualizes a GIC distributor for each guest, preserving the interrupt isolation invariants, (2) handles all other translation faults of the second stage MMU and injects them into the core in guest mode, (3) receives physical interrupts and registers them as virtual interrupts in the GIC, (4) has a hypercall interface to request IGC notification interrupts for outgoing channels of a guest. While the set of handlers is small, the design and verification of the GIC handlers is quite cumbersome, mainly due to the fact that the interrupt controller of our ARMv8 platform is of an older version that does not provide full hardware support for distributor virtualization and distribution of virtual interrupts.

Concerning the modeling of the hypervisor, it would be infeasible to manually specify it on the low abstraction level of our system level semantics. Instead we introduce a high level labeled transition system (LTS) of the hypervisor design, where transitions atomically change (parts of) the system configuration. This kind of reasoning requires an order reduction argument, showing that the fine-grained instruction execution of the hypervisor can be abstracted into atomic blocks [23]. In our case it suffices to design the LTS in such a way that each step contains at most one transition that is either a send or receive action addressing the GIC (which is shared by all cores), or an access to a shared hypervisor data structure. The LTS can be used later as a specification to verify the binary hypervisor code [32].

Formally, we introduce such hypervisor transitions in our system model by adding a new kind of synchronization vector $\textsc{Orcl}\ \omega$, that specifies a system-wide oracle transition according to relation $\omega \subseteq \mathbb{S} \times \mathbb{S}$. In our case ω captures the hypervisor LTS, allowing state changes in one core and its MMU, the SMMUs, and hypervisor data structures in memory, whenever that core is running in hypervisor mode. In the system model we have $\textsc{Orcl}\ \omega \vdash s \to s'$ iff $(s, s') \in \omega$.

4.3 Information Flow Security

In order to show information flow security of the SoC as constrained by the hypervisor we introduce an *ideal model* of the system where each guest is running on an idealized SoC, connected only through IGC channels. If the ideal model is a sound and complete abstraction of the system model, i.e., there exists a bisimulation relation between both, the information flow restrictions that hold in the ideal model *by construction* also hold for the platform model. Then we can use our platform to build trustworthy systems where security-critical services are properly isolated from untrusted software (see Fig.3). In the context of our formal system model, the ideal model is a system where each component is instantiated with one idealized guest SoC.

On this level the only observable message passing is occurring through the IGC notification channels between guests, or via external I/O of one guest SoC. In addition there are oracle transitions that synchronize the contents of memory for the IGC channels between the guests in order to simulate shared memory whenever one of the channels is written to.

The idealized guest SoC models are decomposed systems themselves, similar to the underlying ARMv8 system model, containing only cores and devices of the guest concerned (see Fig.4). The ideal SoC models differ from the platform models on a few important points, namely:

- Ideal cores execute in guest mode only. The hypervisor execution is invisible here and effects of handlers on the core are modeled explicitly as part of the ideal core semantics. For example, hypercall instructions and the reception of memory faults get special semantics that reflect a complete handler execution. Regular core functionality is specified as in the system model. In particular the ideal core model still contains the first stage MMU.

- The range of the ideal memory is restricted to the guest's memory region. (S)MMUs are replaced by simple core and device interface buffers that either forward memory requests to memory if they are within the guest's memory range or produce a fault otherwise. No address translation is performed by these interface buffers, all messages use intermediate physical addresses throughout the ideal model. These placeholders for the (S)MMUs are introduced mainly to simplify the bisimulation proof.

- Each guest SoC has an own ideal GIC with a virtualized distributor and physical interrupt interface. We define the ideal GIC semantics in such a way that it reflects the semantics of the hypervisor handlers that virtualize the distributor and inject virtual interrupts.

- To handle IGC notification interrupts between guests, a special notification interrupt buffer is added for each outgoing channel. The buffers are updated by a hypercall instruction with idealized semantics mirroring the behavior of the underlying hypervisor handler. Upon a global synchonization transition with the receiving guest SoC, an IGC interrupt is injected into the receiver's ideal GIC. This simulates the behavior of the hypervisor receiving the physical inter-processor interrupt and registering it as a virtual interrupt in the GIC.

Note that we use exactly the same device models as in the platform model. This is possible since the hypervisor allocates I/O regions of devices in the intermediate physical address space using an identity mapping, thus devices in both models behave identically.

We prove an invariant $\bar{I}$ on ideal model configurations $\bar{s} \in \bar{\mathbb{S}}$, to support the bisimulation proof and sanity-check our specifications. With initial states $\bar{\mathbb{S}}_0$ defined similar to $\mathbb{S}_0$, we show:

Theorem 2. *Given $\bar{s}_0 \in \bar{\mathbb{S}}_0$, then all computations $\bar{s}_0 \to^n \bar{s}'$ establish $\bar{I}(s')$.*

Given that there is no hypervisor running in the ideal model, the ideal invariant is much simpler, covering basically just well-formedness conditions on the components and messages in each guest. For the memory interface buffers and the ideal GIC it also requires security properties, e.g., that requests sent to memory are within range, or that only interrupts belonging to the guest are pending or active. As initially no requests or interrupts are pending, $\bar{I}(\bar{s}_0)$ holds trivially. In the induction step we first show $\bar{I}$ for internal steps of a guest SoC. Then we prove that for each IGC channel the memory regions in sender and receiver SoC are in sync, if every write into the channel is directly followed by a synchronizing oracle transition.

4.4 Bisimulation Proof

Our final proof goal is to prove a trace equivalence result relating the platform and ideal models. The proof of trace equivalence uses a bisimulation $\mathcal{R} \subseteq \mathbb{S} \times \bar{\mathbb{S}}$ as an unwinding condition. For a platform state s and an ideal state $\bar{s}$, if $s\ \mathcal{R}\ \bar{s}$ then, among others, the following properties on the abstract states of the system components are guaranteed.

- Corresponding cores have the same guest register states and message history variables while the core is running in guest mode.
- Guest memory content in the ideal model is identical to the memory content in the platform model at the translated addresses. Similarly, memory requests and replies are present in the ideal model if, and only if, they are present as guest requests/replies in the platform model at corresponding components with translated addresses. Exceptions are requests sent by cores and devices, which have the same (untranslated) addresses, as well as write requests to and read replies from the virtualized GIC distributor, as they are processed and sent by the hypervisor on behalf of the guest to maintain interrupt isolation.
- The translation table lookups of the (S)MMUs are invisible in the ideal model.
- Device states and message history variables in both models are identical.
- The state of interrupts in the GIC distributor of a guest in the ideal model is the same as in the GIC distributor of the platform model, while no hypervisor interrupt handler is running on one of the guest's cores. Similarly, for each core the same interrupts are pending or active in the ideal virtualized physical interface and the platform virtual interrupt interface.
- IGC interrupts are active in the ideal IGC notification buffer while the corresponding SGI in the platform model is pending.
- The register states of the virtual interrupt interface and the virtualized physical interface are equal. GIC distributor registers are projected to the ideal model using an uninterpreted filtering function that removes for each guest information about the other guests' interrupts.

More complex definitions are needed for the coupling of components, memory messages, and interrupts during the execution of the hypervisor handlers, as the ideal model artifacts have to be linked with the state of the platform during different phases of the hypervisor execution. Since our case study focused on the verification of the system properties guaranteed by the hypervisor, rather than the correctness of the hypervisor implementation itself, we omit further details. The desired correctness theorem can now be stated in two parts.

Theorem 3. *Given initial states $s_0 \in \mathbb{S}_0$ and $\bar{s} \in \bar{\mathbb{S}}_0$ with $s_0\ \mathcal{R}\ \bar{s}_0$, then (1) for any computation $s_0 \rightarrow^n s'$ of the platform model there exists an ideal model computation $\bar{s}_0 \rightarrow^m \bar{s}'$ such that $s'\ \mathcal{R}\ \bar{s}'$ holds, and (2) for any computation $\bar{s}_0 \rightarrow^n \bar{s}'$ of the ideal model there exists a platform model computation $s_0 \rightarrow^m s'$ such that $s'\ \mathcal{R}\ \bar{s}'$ holds.*

Theorem 3 is proved by induction on n for the two directions separately. The base case is trivial. In the induction step we can use the invariants of both models by Theorems 1 and 2. We perform a case split over the different steps of the simulated model, the proof of each case then usually consists of two parts: (1) showing the existence of a – potentially stuttering – corresponding step sequence in the simulating model and (2) showing that resulting states are in the bisimulation relation again, preserving the properties sketched above.

While showing the existence of simulating steps, we often found that our abstract specifications of component behavior based on pre and post conditions were too weak. We had to add verification conditions on the detailed component models, stating that under certain preconditions, derived from relation $\mathcal{R}$ and the invariants, the corresponding transition is indeed enabled. Assuming these proof obligations, the existence argument is usually straightforward.

Whenever the hypervisor handlers are not involved, steps of both models are mostly mapped in a one-to-one fashion. An exception are the translation steps of the (S)MMUs that are invisible in the ideal model. When performing the simulation of a core or device sending a request to its memory interface buffer in the ideal model, we step the (S)MMU until either a fault occurs or the translation is successful, using the (S)MMUs' progress condition. Similarly, ideal transitions that are implemented by a handler are simulated by executing the handler in its entirety.

We add an induction hypothesis to the simulation of the ideal by the platform model reflecting our stepping strategy. It states, e.g., that cores of the platform model are always in guest mode and that no requests are currently being translated. This saves us from starting the simulation of an ideal transition in the middle of a hypervisor handler or an address translation in the MMU. In the opposite simulation direction we cover all these intermediate states.

When proving the bisimulation relation, we profit from the compositional approach, as at most two components change in one model. Then clauses of $\mathcal{R}$ relating other components are usually preserved trivially. Additional verification conditions on the detailed component models are also needed when stepping cores or the GIC. As our abstract specification leaves the detailed effects of transitions on registers and internal states undefined, it is impossible to derive that the same transition on related components has the same effect on both, especially when transitions are non-deterministic. Therefore, we introduce proof obligations on the core and GIC transitions for these cases, demanding just that: if two components are coupled according to the bisimulation relation and one of them performs a particular transition, then there exists a corresponding transition on the other component such that the coupling is preserved.

Naturally, it is easy to make too strong assumptions here, hence it is crucial to discharge them on the detailed component models. If such a proof fails, the abstract component specifications or the bisimulation relation for that component need to be strengthened. Here the decomposed verification approach reduces the cost of re-verification, too. Nevertheless, the bisimulation proof guarantees that all detailed component instantiations, satisfying the verification conditions, exhibit the same secure information flow as the ideal model.

4.5 Implementation

We have modeled our case study in the theorem prover HOL4 and proved central obligations of the guest execution (i.e. transitions not concerning the hypervisor). Further verification is ongoing. The ideal invariant is verified while the platform invariant is only proven on paper so far. The induction step for the bisimulation property comprises 10 hypervisor-transitions and 43 transitions that at least partly concern guest execution. Currently, we have verified about a third of the latter. The successfully verified obligations include the most challenging transitions, for instance those including address translation and the virtual interrupt interface, and cover transitions of all SoC components in at least one direction of simulation. Remaining challenges regarding guest execution mostly concern interrupt signaling to the cores. Besides those cases, the remaining proofs should be straight-forward adaptations of the existing ones for similar cases. Table 1 provides an overview on the size of the different parts of our development. We consider "basic" parts and the machinery as reusable for similar endeavors. The case study so far took about 14 person months with low to intermediate level of a priori HOL4-expertise.

	basic	common	ideal	platform	hyperv.	bisim.	total
model specification	99	435	1,121	1,750	1,440	350	5,195
invariant specification	–	17	387	518	–	453	1,375
machinery	309	–	95	–	–	585	989
proofs	652	1,094	1,132	1,466	145	7,437	11,926
total	1,060	1,546	2,735	3,734	1,585	8,825	19,485

Table 1: HOL4 lines of code for basic parts (general data types etc.), common parts for both models, the ideal model, the platform model, the hypervisor, and the bisimulation proof.

We also started to verify hypervisor transitions, but decided to increase the support for automation first before tackling the simulation of handler step sequences. To date, we have developed some first machinery to step through bisimulation-proofs that map a transition step of one side to a number of transition steps on the other side. The ambition is that the human proof engineer should be able to focus on one transition step at the time and in particular on the actual preservation property, leaving (de-)composition and rather trivial proof obligations to the machinery. To that end, we employ a canonical form for bisimulation goals that we reestablish between the different steps. The machinery is able to autonomously identify relevant pre and post states, as well as transitions, relations and guarantees on them, with the help of some form of incremental pattern matching. Several custom tactics automate unfolding and employ tailored variants of standard machinery such as first order reasoning, conditional lifting, or simplifications for record field updates, case splits, etc. For future work we plan to extend the machinery by the automatic identification and verification of linking pre and post conditions.

A large part of the proofs is concerned with technical arguments about the transition system, e.g., the matching of memory requests and replies, the relation of different address regions, or the well-formedness of system parameters. We envision a formal framework for decomposed system models that provides a lot of these properties by construction for a given instantiation and avoids the pitfalls discovered in our ad-hoc definition.

5 Conclusion

With the increasing hardware support for virtualization and enhanced security threats through malicious devices, security analysis of embedded systems today needs to consider the behavior of all SoC components. To address this issue, we presented an approach for the compositional verification of SoC level security properties in a virtualization context. It enables a top-down approach to system verification, by modeling SoC components as communicating automata with relatively abstract specifications, that can be refined gradually into more detailed component models. Our HOL4 case study for an ARMv8 hypervisor highlighted the reusability and adaptability of the approach. The preliminary results suggest that the verification of security properties for a complete SoC is feasible, yet still time-consuming, especially for complex COTS systems. For future work we therefore envision a formal framework supporting both the modeling (e.g. with a domain specific language) and the reasoning (with more automation), something for which HOL4 seems particularly well suited. The discussed case study – which we are going to progress further – provided valuable insights towards that goal. Further directions of improvement include discharging proof obligations discovered in the proof for the existing detailed component models of the ARMv8 core and MMU, demonstrating property transfer between bisimilar decomposed models, as well as supporting more realistic memory models.

6 Acknowledgments

We thank Thomas Tuerk for his support in improving our HOL4 definitions. Work supported by the PROSPER project funded by the Swedish Foundation for Strategic Research, the HASPOC project funded by the Swedish Innovation Agency VINNOVA, and the KTH CERCES Center for Resilient Critical Infrastructures funded by the Swedish Civil Contingencies Agency.

References

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[2] Martín Abadi and Leslie Lamport. An old-fashioned recipe for real time. *ACM Trans. Program. Lang. Syst.*, 16(5):1543–1571, September 1994.

[3] Eyad Alkassar, Mark A. Hillebrand, Wolfgang J. Paul, and Elena Petrova. Automated verification of a small hypervisor. In *Proc. VSTTE*, volume 6217 of *LNCS*, pages 40–54. Springer, 2010.

[4] Rajeev Alur, Thao Dang, Joel Esposito, Rafael Fierro, Yerang Hur, F Ivančić, Vijay Kumar, Insup Lee, Pradyumna Mishra, George Pappas, and O. Sokolsky. Hierarchical hybrid modeling of embedded systems. In *Embedded Software (EMSOFT)*, pages 14–31. Springer, 2001.

[5] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, and Carlos Luna. Formally verifying isolation and availability in an idealized model of virtualization. In *Formal Methods*, pages 231–245, 2011.

[6] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, and Carlos Luna. Cache-leakage resilient os isolation in an idealized model of virtualization. In *Proc. CSF'12*, pages 186–197. IEEE, 2012.

[7] Christoph Baumann, Mats Näslund, Christian Gehrmann, Oliver Schwarz, and Hans Thorsen. A high assurance virtualization platform for ARMv8. In *European Conference on Networks and Communications (EuCNC)*, pages 210–214, June 2016.

[8] Pauline Bolignano, Thomas Jensen, and Vincent Siles. Modeling and abstraction of memory management in a hypervisor. In *FASE/ETAPS*, pages 214–230. Springer, 2016.

[9] Hao Chen, Xiongnan (Newman) Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. Toward compositional verification of interruptible OS kernels and device drivers. In *Proceedings of Programming Language Design and Implementation*, PLDI '16, pages 431–447. ACM, 2016.

[10] Alessandro Cimatti and Stefano Tonetta. Contracts-refinement proof system for component-based embedded systems. *Science of computer programming*, 97:333–348, 2015.

[11] Michael R Clarkson and Fred B Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[12] Mads Dam, Roberto Guanciale, Narges Khakpour, Hamed Nemati, and Oliver Schwarz. Formal verification of information flow security for a simple ARM-based separation kernel. In *Proceedings of Computer and Communications Security*, CCS'13, pages 223–234. ACM, 2013.

[13] Richard J Feiertag and Peter G Neumann. The foundations of a provably secure operating system (PSOS). In *National Computer Conference*, pages 329–334. AFIPS Press, 1979.

[14] Shaked Flur, Kathryn E Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. *ACM SIGPLAN Notices*, 51(1):608–621, January 2016.

[15] Anthony C. J. Fox. Improved tool support for machine-code decompilation in HOL4. In *Interactive Theorem Proving (ITP)*, pages 187–202, 2015.

[16] D. D. Gajski and F. Vahid. Specification and design of embedded hardware-software systems. *IEEE Design Test of Computers*, 12(1):53–67, 1995.

[17] Liang Gu, Alexander Vaynberg, Bryan Ford, Zhong Shao, and David Costanzo. CertiKOS: a certified kernel for secure cloud computing. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys'11, page 3. ACM, 2011.

[18] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. Cache storage channels: Alias-driven attacks and verified countermeasures. In *Security and Privacy*, pages 38–55, 2016.

[19] HASPOC project. http://haspoc.sics.se/.

[20] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In *Operating Systems Design and Implementation*, pages 165–181. USENIX Association, 2014.

[21] Nannan He, Daniel Kroening, Thomas Wahl, Kung-Kiu Lau, Faris Taweel, Cuong Tran, Philipp Rümmer, and Sanjiv Sharma. Component-based design and verification in X-MAN. *Proc. Embedded Real Time Software and Systems*, 2012.

[22] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby C. Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.*, 32(1):2, February 2014.

[23] Richard J Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

[24] Hamed Nemati, Roberto Guanciale, and Mads Dam. Trustworthy virtualization of the ARMv7 memory subsystem. In *SOFSEM*, pages 578–589. Springer, 2015.

[25] Wolfgang J. Paul, Sabine Schmaltz, and Andrey Shadrin. Completing the automated verification of a small hypervisor - assembler code verification. In *SEFM*, volume 7504 of *Lecture Notes in Computer Science*, pages 188–202. Springer, 2012.

[26] Alastair Reid. Trustworthy specifications of ARMv8-A and v8-M system level architecture. In *Proceedings of Formal Methods in Computer-Aided Design (FMCAD)*, 2016.

[27] James A. Rowson and Alberto Sangiovanni-Vincentelli. Interface-based design. In *Proceedings of the 34th Annual Design Automation Conference*, DAC '97, pages 178–183. ACM, 1997.

[28] Fernand Lone Sang, Eric Lacombe, Vincent Nicomette, and Yves Deswarte. Exploiting an I/OMMU vulnerability. In *Malicious and Unwanted Software (MALWARE)*, pages 7–14, 2010.

[29] Oliver Schwarz and Mads Dam. Formal verification of secure user mode device execution with DMA. In *Hardware and Software: Verification and Testing (HVC)*, number 8855 in Lecture Notes in Computer Science, pages 236–251, 2014.

[30] Oliver Schwarz and Mads Dam. Automatic derivation of platform noninterference properties. In *Software Engineering and Formal Methods*, pages 27–44. Springer, 2016.

[31] Peter Sewell and Jan Vitek. Secure composition of insecure components. In *Computer Security Foundations*, CSFW '99, pages 136–. IEEE Computer Society, 1999.

[32] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *Programming Language Design and Implementation*, pages 471–482, 2013.

[33] Patrick Stewin and Iurii Bystrov. Understanding DMA malware. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 21–41, 2012.

[34] Hira Syeda and Gerwin Klein. Reasoning about translation lookaside buffers. In *Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 490–508, may 2017.

[35] Amit Vasudevan, Sagar Chaki, Petros Maniatis, Limin Jia, and Anupam Datta. überSpark: Enforcing verifiable object abstractions for automated compositional security analysis of a hypervisor. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, 2016.

[36] Rafal Wojtczuk. Subverting the Xen hypervisor. *Black Hat USA*, 2008.

[37] Fei Xie, Guowu Yang, and Xiaoyu Song. Component-based hardware/software co-verification for building trustworthy embedded systems. *J. Syst. Softw.*, 80(5):643–654, May 2007.