Journal of Cryptographic Engineering (2019) 9:243–261
https://doi.org/10.1007/s13389-019-00216-4

REGULAR PAPER

On the verification of system-level information flow properties for virtualized execution platforms

Christoph Baumann[1] · Oliver Schwarz[2] · Mads Dam[3]

Received: 31 January 2018 / Accepted: 14 May 2019 / Published online: 25 May 2019

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 and report on the complete verification of guest mode security properties in the HOL4 theorem prover.

Keywords Decomposition · SoC · Information flow security · Formal verification · Hypervisor · ARMv8

1 Introduction

The rise of embedded systems and the internet of things has been met 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 toward 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 [23] 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 [57]) might be able to circumvent the kernel's memory isolation and install stealthy key loggers [51] or even take control of the whole system [38]. Input/output memory management units (IOMMUs/SMMUs) allow the kernel to constrain the address ranges accessible by devices. However, IOMMUs are not always free from vulnerabilities either [45] 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, the size and complexity of current generation SoCs and the general unavailability of authoritative system-level formal models[1] make a comprehensive security verification a highly challenging task. In addition, the cost-efficiency of system-level verification is adversely affected by the heterogeneous nature of and steady evolution of SoC designs.

✉ Christoph Baumann
christoph.baumann@ericsson.com

Mads Dam
mfd@kth.se

[1] Ericsson Research Security, Kista, Sweden

[2] RISE SICS, Kista, Sweden

[3] KTH Royal Institute of Technology, Stockholm, Sweden

[1] The open-source RISC-V architecture [42] and the recent machine-readable specifications released by ARM [41] are notable exceptions to this.

In this paper we report on a tool-assisted experiment using the HOL4 theorem prover to verify information flow security for an industry-scale security-oriented bare-metal hypervisor on ARMv8 [8]. Developed in the open-source HASPOC project [25], the hypervisor provides full virtualization with low performance overhead 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 partitions system resources, i.e., cores, memory, devices, and interrupts between the guests; thus, there is no resource sharing, except through 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, (S)MMUs, cores—including their user mode capabilities—as well as interrupt controllers.

To verify, even partially, a system of this complexity requires extensive use of composition and abstraction. The SoC is modeled as a network of distributed automata that communicate using 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 are especially important for embedded systems, since standard chip sets are rare and verification needs to be performed for many different custom SoC designs.

The top goal of the verification 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 may break the isolation guarantees imposed on the system.

The security property is formulated as trace equivalence between an ARMv8 platform model and an idealized specification where guest systems are running on dedicated SoCs with explicit communication channels between them, in the style of [15]. Both models and the hypervisor design are completely formalized in the HOL4 theorem prover and based on the user-level ARMv8 CPU model of Fox [17]. We show trace equivalence using a bisimulation between the ideal and platform model as an unwinding condition. While verification work is ongoing, for the part of the bisimulation covering guest execution, i.e., the steps not virtualized by hypervisor handlers, all corresponding cases have been verified with reasonable effort. The formal artifacts are available online along with detailed technical documentation of the models and theorems involved [9].

The work reported here pushes forward the state of art in several directions. Previous work, e.g., seL4 [32], CertiKOS [22], or Verisoft [39], has focused on core level execution only with limited attention to system-wide aspects such as interrupt processing, and in this context little attention has so far been paid to the formal analysis of information flow properties. Specifically, the paper makes the following contributions:

- A system model, formalized in HOL4, based on an earlier L3/HOL4-based model by Fox [17], of a multi-core ARMv8-based SoC, including virtualization and TrustZone extensions, two-stage MMUs, SMMUs, devices, and an ARM-based Generic Interrupt Controller (GIC).
- An ideal hypervisor model expressing the desired isolation properties of a hypervisor in terms of physically separated “virtual” SoCs, connected through an interrupt interface and shared channel buffers in memory. The design and requirements of the hypervisor itself are described in more detail in [8].
- A proof of trace equivalence between the system model and the ideal model, with key parts machine-verified using the HOL4 theorem prover.
- A demonstration how the trace equivalence result can be used to transfer security properties at the ideal model level to the system model.
- The development of key abstraction-based mechanisms that help reduce the amount of detail needing to be considered in the formal proof.

The paper is organized as follows: After discussing related work in Sect. 2 we present the high-level modeling framework and the general abstraction mechanisms used to realize the proofs in Sects. 3 and 4. We detail the ARMv8-based platform model in Sect. 5 and the hypervisor model in Sect. 6. The ideal model is introduced in Sect. 7, and in Sect. 8 we present the trace equivalence proof. Then, in Sect. 9 we show how the result can be used to derive information flow security. In Sect. 10 we discuss the implementation of the proof in the HOL4 theorem prover. Finally, in Sects. 11 and 12 we discuss some of the issues, design choices, and limitations encountered during this work, as well as directions for future research.

2 Related work

The merits of compositional reasoning for the design and validation of embedded systems have been widely acknowledged in the field, c.f. [5,18,27,49,58]. A compositional approach based on component abstraction and rely-guarantee reasoning is also at the core of the contract-based design and verification paradigm [14,43]. In this work, we apply the same underlying techniques to the formal verification of security properties for the low-level execution platform, which 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 [16]. The research discipline has gained increased traction in recent years through prominent projects such as seL4 [32] and Verisoft (XT) [4,39]. 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 two-stage MMUs. Given the central role of memory management, recent work modeled the effects of several kinds of MMUs, their proper configuration, caches, TLBs, and their interplay with system software [6,7,12,37,52]. 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 [13]. Kernel verification has been studied both for settings with IOMMUs [21,26,54] and for peripherals configured to comply with constrained access policies [46]. 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 in unforeseen ways, as the Meltdown [34], Spectre [33], and Foreshadow [55] attacks demonstrate quite strikingly. While it is possible to mitigate such threats, it is important to be aware of them both in the design and verification of kernels through the use of precise hardware models. ARM started to create machine-readable ISA specifications [41] 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 [47]. Arguably, these models are still not detailed enough to capture all possible system security attack vectors. Here, instead of focusing on single system parts at varying levels of detail, we are interested in a holistic view and system-wide isolation guarantees, where existing results can potentially be reused as building blocks for models that may gradually be refined to capture all possible behaviors of the underlying hardware.

3 System model

Systems are modeled as a (finite) collection $\mathcal{K}$ of components $\mathcal{K}(i)$, where i is a member of some index set I. A component is a labeled transition system with states $\sigma \in \Sigma$, initial states $\Sigma_0 \subseteq \Sigma$, and transitions $\mathrm{snd}(\sigma, m, \sigma')$, $\mathrm{rcv}(\sigma, m, \sigma')$, and $\tau(\sigma, \sigma')$ representing, respectively, the sending and receiving of message m (the nature of which is left unspecified for now), and the internal, unobservable transition from component state σ to component state σ'. We use relations and not functions to represent transitions, in order to not rule out nondeterministic component models.

$$\frac{\begin{array}{cc} i,j \in I & i \neq j \\ \mathcal{K}(i).\mathrm{snd}(s(i), m, s'_i) & \mathcal{K}(j).\mathrm{rcv}(s(j), m, s'_j) \end{array}}{\textsc{Msg}\ i\, j\, m \vdash s \rightarrow s[i \mapsto s'_i; j \mapsto s'_j]}\ \textsc{Msg}$$

$$\frac{i \in I \qquad \mathcal{K}(i).\tau(s(i), s'_i)}{\textsc{Tau}\ i \vdash s \rightarrow s[i \mapsto s'_i]}\ \textsc{Tau}$$

$$\frac{i \in I \qquad \mathcal{K}(i).\mathrm{rcv}(s(i), m, s'_i)}{\textsc{ExtI}\ i\, m \vdash s \rightarrow s[i \mapsto s'_i]}\ \textsc{ExtI}$$

$$\frac{i \in I \qquad \mathcal{K}(i).\mathrm{snd}(s(i), m, s'_i)}{\textsc{ExtO}\ i\, m \vdash s \rightarrow s[i \mapsto s'_i]}\ \textsc{ExtO}$$

Fig. 1 Overall system semantics. Component i in state s is updated to s'_i using notation $s[i \mapsto s'_i]$

Components synchronize between themselves using synchronization vectors $t \in \mathcal{T}$ defined by:

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

where $i, j \in I$. 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 $\textsc{ExtO}\ i\ m$ denote external I/O actions of component i with associated messages m from or to the system's environment.

A (global) system state $s \in \mathbb{S}$ is now a mapping from component indices $i \in I$ 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 $t \vdash s \rightarrow s'$ 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. We overload our transition notation for schedules $t = t_1, \ldots, t_n \in \mathcal{T}^*$, i.e., $t \vdash s \rightarrow s'$ means that s' can be reached from s by executing the transitions t_j in order for $j \in [1 : n]$. For the empty schedule ε we set $\varepsilon \vdash s \rightarrow s' \equiv (s = s')$.

4 Modeling and verification approach

Below we discuss the instantiation of the system model for a given system-on-chip and introduce our abstraction-based verification methodology.

4.1 Modeling a system-on-chip

When instantiating the above framework to a concrete SoC, we generally model bus masters, e.g., cores, devices, and the memory system, including caches and RAM, as separate components, and the memory buses and interrupt signals as synchronization vectors. External inputs and outputs are specific for each device, modeling, for instance, network packages, user input, sensor data, or actuator control signals.

Communication through memory buses and interrupt signals is by its nature mostly asynchronous, i.e., a sender does not usually know when a message will be received or a potential reply be returned. In order to map such asynchronous communication to the synchronous rendezvous of the modeling framework introduced above, the channels can be modeled as buffers, either separate or merged with the sending or receiving component. 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. A component that both receives and sends information simultaneously can 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 transitions represent atomic actions in the SoC model. When defining the component transitions, care must be taken wrt. atomicity not to rule out interleavings of actions that may lead to externally observable behavior. To this end, internal transitions can be used to control the granularity of component transitions and expose intermediate component states that are observable by other components or the environment through communication.

The instantiation proposed above does not explicitly model time. As computer systems are discrete, and external communication is generally asynchronous, the exact timing of signals and component transitions can often be neglected so that the interleaving model captures all system behaviors. In principle, time can be represented as an additional variable of the model [1] and by restricting possible interleavings accordingly. In practice, however, current multi-core commodity platforms like ARMv8 are generally regarded as too complex for precise time modeling to be feasible [56].

4.2 Abstraction

Proving properties of an execution platform of ARMv8's complexity requires very extensive use of abstraction to not get overwhelmed by detail. In the work reported here, we have used a simple abstraction mechanism that in many cases allows us to greatly reduce the amount of information needing to be processed. Formally, we introduce abstract component states $\hat{\Sigma}_i$, and abstraction (aka lifting) functions $\lceil\cdot\rceil_i : \mathcal{K}(i).\Sigma \rightarrow \hat{\Sigma}_i$ for all $i \in I$. For example, a memory M may be implemented using a complex hierarchy of storage elements, such as caches, RAM, and nonvolatile memory. An abstraction $\lceil M \rceil_i$ that hides these details from a top-level proof may instead view the memory contents as a flat map of addresses to values, when doing so is adequate to establish the desired result.

We refrain from also lifting the transitions. Instead, we introduce a notion of *abstract transition specification* that constrains the behavior of the concrete components via predicates on the abstract states. The specifications act as proof obligations that need to be discharged for any instantiation of a component covered by the abstraction, and can thus be used in place of a concrete component model in a top-level proof. We use two types of such specifications. The first is a *behavioral specification* that restricts the semantics of a component transition using a condition P on the abstract prestate and a condition Q to relate the abstract pre- and poststates. For all transitions sending a message m (and similarly for receiving and internal steps), we require the following property, given a behavioral specification (P, Q) for component i:

$$\begin{aligned}\forall \sigma, m, \sigma'.\ \mathcal{K}(i).\mathrm{snd}(\sigma, m, \sigma') \ \Rightarrow\ & P(\lceil\sigma\rceil_i, m) \\ & \wedge\ Q(\lceil\sigma\rceil_i, m, \lceil\sigma'\rceil_i)\end{aligned}$$

That is, the pre- and postconditions P and Q must be sound in the sense that if a transition is taken, then the precondition holds on the abstract prestate, and the postcondition correctly links the abstract prestate with the abstract poststate, possibly conditional on some system-level invariant established separately. As an example of a behavioral specification, consider a memory that receives and answers read requests. In the latter case a necessary precondition (P) is that any answer m sent by the memory is related to a request pending in the prestate. Postcondition Q would at least require that the value being sent is consistent with the content of the requested address in the abstract memory view and that memory contents are unchanged. Note that, P and Q implicitly induce a transition relation on the abstract states for messages m. The required property then effectively states that this relation is a sound abstraction of the detailed component semantics.

Secondly, an *enabling specification* gives a sufficient condition E on an abstract prestate under which a certain concrete component transition is enabled. In its simplest form, we require the following property for a transition out of a prestate σ, sending message m.

$$\forall \sigma, m.\ E(\lceil\sigma\rceil_i, m) \ \Rightarrow\ \exists \sigma'.\ \mathcal{K}(i).\mathrm{snd}(\sigma, m, \sigma')$$

In our memory example, E could, for instance, say that a matching request for answer m is pending and that the value in m matches the memory contents in $\lceil\sigma\rceil_i$. Then, the specification would require that the memory is always able to send this answer. We also allow to existentially quantify over the sent message m, e.g., saying that for every read request pending, an answer matching the memory contents may be sent. In a more general form, enabling specifications are pairs of predicates (E, F), requiring the following property for sending transitions (and similarly for rcv and τ) of a component i:

$$\begin{aligned}\forall \sigma, x.\ E(\lceil \sigma \rceil_i, x) \ \Rightarrow \ \exists \sigma', m.\, \mathcal{K}(i).\mathrm{snd}(\sigma, m, \sigma') \\ \wedge\, F(x, \lceil \sigma \rceil_i, m, \lceil \sigma' \rceil_i) \qquad (1)\end{aligned}$$

Here, we introduce a symbolic variable x to capture any information pertaining to prestate σ that is used by F to constrain the desired transition. For instance, in the example above, x would represent any pending read request in σ and F would demand that m matches x and the contents of σ. In general, we allow also to demand certain effects of the required transition by constraining its abstract poststate $\lceil \sigma' \rceil$ via F. Observe that the simple enabling condition above is an instance of the general form with $x = m$ and $F(x, \hat{\sigma}, m', \hat{\sigma}') \equiv (x = m')$.

A typical application in the verification reported below (cf. Sect. 8) is to establish a bisimulation relation $\mathcal{R}$ between a high-level ideal model and a refined model reflecting the behavior realized in the concrete system. Then, if we use $\overline{\sigma}, \overline{\sigma}'$ to represent ideal states and σ, σ' to represent states in the refined model, then a typical verification condition using abstractions to aid modeling at both ideal and refined model levels might look like the following:

$$\begin{aligned}&\forall \overline{\sigma}, \sigma, m, \sigma'.\ \lceil \sigma \rceil_i \ \mathcal{R}\ \lceil \overline{\sigma} \rceil_i \wedge \mathrm{snd}(\sigma, m, \sigma') \\ &\quad \Rightarrow \exists \overline{\sigma}'.\ \mathrm{snd}(\overline{\sigma}, m, \overline{\sigma}') \wedge \lceil \sigma' \rceil_i \ \mathcal{R}\ \lceil \overline{\sigma}' \rceil_i\end{aligned} \qquad (2)$$

In this case the enabling specification (1) is instantiated by formula (2) as follows:

- $x = (\sigma, m, \sigma')$,
- $E(\lceil \overline{\sigma} \rceil_i, x) = \lceil \sigma \rceil_i \ \mathcal{R}\ \lceil \overline{\sigma} \rceil_i \wedge \mathrm{snd}(\sigma, m, \sigma')$, and
- $F(x, \lceil \overline{\sigma} \rceil_i, m', \lceil \overline{\sigma}' \rceil_i) = \lceil \sigma' \rceil_i \ \mathcal{R}\ \lceil \overline{\sigma}' \rceil_i \wedge m = m'$,

Behavioral and enabling specifications are not independent. In particular, the sufficient preconditions of a transition need to imply the necessary ones; otherwise, the specifications could contradict each other and we would assume specifications that are logically equivalent to *false*. For the same reason, also the predicates F, P, and Q must not be contradictory. Formally, we expect specifications to pass the following sanity checks.

$$\begin{aligned}&\forall \hat{\sigma}, x, m, \hat{\sigma}'.\ E(\hat{\sigma}, x) \wedge F(x, \hat{\sigma}, m, \hat{\sigma}') \ \Rightarrow \ P(\hat{\sigma}, m) \\ &\quad \forall x.\ \exists \hat{\sigma}, m, \hat{\sigma}'.\ F(x, \hat{\sigma}, m, \hat{\sigma}') \wedge P(\hat{\sigma}, m) \wedge Q(\hat{\sigma}, m, \hat{\sigma}')\end{aligned}$$

The abstraction technique outlined above allows to start verification with an underspecified abstract model which passes the sanity checks and only later develop a more detailed model that concretizes the abstract one, reducing the initial modeling effort. This approach can also be applied to the definition of the messages of the system, i.e., leaving certain message parts underspecified when they are not essential to the proof. In practice, for our case study we have experienced that component specifications can be made very abstract using uninterpreted and underspecified types and functions, without losing expressiveness wrt. the verification of platform security properties. During instantiation one needs to fill in these gaps, i.e., make the definitions of the abstract states, the messages, and the abstraction function precise. Also, the proof obligations stated as abstract transition specifications must be discharged.

An important question is then how detailed the abstraction should be. There are several considerations. An important one is instantiation and the corresponding proof obligations: The abstraction must be detailed enough to identify when a given transition is enabled. Also the abstraction needs to expose all state variables required to express a desired property. In the area of information flow security, this usually means that all state variables directly or indirectly observable by an attacker must be represented at the abstract level. For a given abstraction $\lceil \cdot \rceil_i$, this requirement is expressed by the following *completeness proof obligation* in case of a sending transition (and similarly in the other cases).

$$\begin{aligned}&\forall \sigma_2, \sigma_1, m, \sigma_1'.\ \lceil \sigma_1 \rceil_i = \lceil \sigma_2 \rceil_i \wedge \mathcal{K}(i).\mathrm{snd}(\sigma_1, m, \sigma_1') \\ &\quad \Rightarrow \exists \sigma_2'.\ \mathcal{K}(i).\mathrm{snd}(\sigma_2, m, \sigma_2') \wedge \lceil \sigma_1' \rceil_i = \lceil \sigma_2' \rceil_i\end{aligned}$$

Again the completeness proof obligation is an enabling specification for $x = (\sigma_1, m, \sigma_1')$, $E(\lceil \sigma_2 \rceil_i, x)$ being defined like the antecedent of the implication, and

$$F(x, \lceil \sigma_2 \rceil_i, m_2, \lceil \sigma_2' \rceil_i) \equiv \lceil \sigma_1' \rceil_i = \lceil \sigma_2' \rceil_i \wedge m_2 = m\ .$$

Intuitively, if an abstraction does not fulfill the property for a given instantiation, it means there is a variable in the instantiation that is not covered by the abstraction function, but its values leak directly or indirectly into other abstract state variables. Hence, for such an instantiation the corresponding abstraction is too weak to be used in arguments about component i's information flow properties. Nevertheless, it can still be used to prove safety properties of any valid instantiation.

5 ARMv8 platform model

As the basis for our modeling work, we extended the user-level ARMv8 CPU model by Fox [17] 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. For a

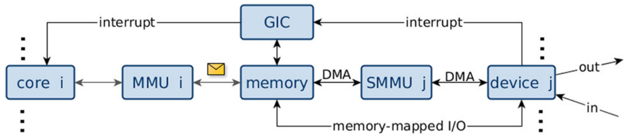

Fig. 2 Decomposed ARMv8 platform model: arrows show the possible flow of messages between different components; other cores, (S)MMUs, and devices in the system are elided

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.

The full model decomposes the SoC state $s \in \mathbb{S}$ into the following components:

- a parameterized number of ARMv8 cores including their first-stage MMUs,
- the corresponding second-stage MMU for each core,
- a shared flat main memory component,
- a parameterized number of arbitrary devices,
- a corresponding SMMU component for each device,
- 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 core models. In a hypervisor scenario the first-stage MMUs are controlled by the untrusted guests; therefore, their interactions with the cores are irrelevant to overall system security allowing to simplify the model. For this reason, we refer to second-stage MMUs simply by "MMU". Also, having one SMMU per device may seem like a strong assumption, but modern SMMUs usually manage different session IDs for different devices; hence, we model them as private SMMUs for each device.[2]

The possible communication channels between the components, i.e., the synchronization vectors, are depicted in Fig. 2. We distinguish

- memory requests ($\mathbb{Q}$) and replies ($\mathbb{R}$),
- virtual and physical interrupts, and
- external input and output signals for the devices,

as messages of our system model. Memory requests are reads, writes, or page table walks of some (S)MMU, and they record at least the address and size of memory accesses as well as the written values in case of writes. In addition, requests contain an uninterpreted component to encode further instantiation-dependent message information, e.g., memory access types, cacheability attributes, or unique message identifiers.

Memory replies contain either a result for a matching request or a fault, e.g., due to failed MMU access permission checks. Our case study does 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 (cf. Fig. 2). Similarly, devices access the memory directly (DMA) 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 configuration of the GIC distributor module. Cores can also request software-generated interrupts (SGIs) to other cores through MMIO accesses to the GIC distributor, and the hypervisor can configure virtual interrupts for the guests through accesses to the corresponding GIC virtualization control interface.

The component configurations in our system model are uninstantiated, and we describe their behavior instead using abstract specifications, 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. We distinguish different transitions of components depending on the type of the transition (send, receive, internal) and the type of message involved (memory, interrupt, external). For each transition, we provide a behavioral specification, capturing pre- and postconditions of the transition, and an enabling specification, capturing when a specific transition is enabled. Almost all components can perform internal transitions, which are mostly underspecified except for history variable updates. Further details on the components and their specifications are given below.

5.1 Core and first-stage MMU

We model explicitly the program counter (PC) and processor status register (containing among others the execution level). The remaining register state is split 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 registers controlling the first-stage MMU. The hypervisor register state contains control registers accessible in EL2 and EL3, and we only model those relevant for the

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

hypervisor design, e.g., HCR_EL2 and SCR_EL3, controlling traps from guest mode.

In addition to the registers and the history variable recording outstanding memory requests, the abstract core state also contains an uninterpreted instantiation-dependent state variable that is used to represent internal core information like the currently executing instruction(s) or the pipeline state. Since it is uninterpreted, we can only reason about its semantics using equality, i.e., stating that two internal states are equal or not, or by introducing additional abstractions on top of it. The internal state variable determines, e.g., when the cores are ready to send memory requests, and we make use of it when coupling cores in the platform model with corresponding cores in the idealized system model, requiring that they have the same internal state during guest execution.

The possible transitions of the core are sending or receiving a memory request and receiving a virtual or physical interrupt in addition to internal transitions representing, e.g., arithmetic operations. We leave transitions that do not change the execution mode 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. Note that, the enabling specification for sending requests cannot be given directly, because it depends on the internal state of the core. Instead, we provide a specification in relation to an ideal model core, which is explained later.

Memory replies may be received from the MMU if there is a matching outstanding request. They are modeled similarly to send transitions unless a fault is received. Then, on system calls, 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. For instance, a behavioral specification of receiving an asynchronous interrupt requires the necessary precondition that interrupts are not masked and no outstanding memory request is waiting for an answer. Only then is the exception taken, setting the PC to the corresponding interrupt vector, saving the guest context to banked registers, and changing the mode to EL2, among other effects prescribed by the postcondition.

5.2 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 (i.e., registers and page tables) as an abstract translation scheme parameterized for each guest and by keeping track of the translation status for pending memory requests.

To give a flavor of our abstraction method, we define the abstract state of an MMU below. It contains the following variables:

- $\mathit{active} \in \mathbb{B}$ Denotes whether the second-stage MMU is enabled (always *true* when guest is running),
- $\mathit{QR}, \mathit{QS} \subset \mathbb{Q}$ History variables recording the outstanding memory requests received from the cores and sent to memory (including page table lookups and translated requests),
- $\mathit{RR}, \mathit{PTL} \subset \mathbb{R}$ History variables recording received memory replies that need to be forwarded to the core and the accumulated set of page table lookup memory replies,
- *cfg* The uninterpreted configuration of the MMU, defining, e.g., the page table base or translation mode (constant after system initialization),
- $\mathit{state} : \mathbb{Q} \to \bot \mid \textsc{Trans}\ \mathbb{Q} \cup \bot \mid \textsc{Final}\ \mathbb{Q} \cup \bot \mid \textsc{Fault}$ The current translation status of core requests to the MMU. Status $\mathit{state}(q) = \bot$ means that q is not currently pending at the MMU, $\textsc{Trans}\ q'$ means that q is being translated and a page lookup q' was requested from memory, and $\textsc{Final}\ q'$ means that q was translated into q' and sent to memory. In both cases $q' = \bot$ is used if a corresponding memory request is yet to be sent. Naturally, status $\textsc{Fault}$ indicates a translation fault, e.g., due to lacking permissions.

While the hypervisor is running on a core, we model its MMU as being turned off and core requests are just forwarded to memory without translation. If an MMU u is enabled, i.e., $\lceil u \rceil.\mathit{active}$, we have the following cases and behavioral specifications for transitions to u':

Receiving a core's memory request q We require that q is not already pending, i.e., $q \notin \lceil u \rceil.\mathit{QR}$. No abstract state variables change except $\lceil u' \rceil.\mathit{QR} = \lceil u \rceil.\mathit{QR} \cup \{q\}$ and $\lceil u' \rceil.\mathit{state}(q) \in \{\textsc{Trans}\ \bot, \textsc{Fault}\}$, i.e., q is recorded in the received requests and either enters the translation phase or it fails directly, for instance, due to an address size error.

Sending a translation table lookup or a final memory request q′ to memory In either case $q' \notin \lceil u \rceil.\mathit{QS}$ and there must not be a reply matching q' in $\lceil u \rceil.\mathit{RR}$ already. Moreover, there needs to exist a pending request q with $\lceil u \rceil.\mathit{state}(q) \in \{\textsc{Trans}\ \bot, \textsc{Final}\ \bot\}$ for which q' is sent. The next state for q is then either $\textsc{Trans}\ q'$ and q' is a page table lookup, or $\textsc{Final}\ q'$ and q' is an address-translated version of q. Nothing else changes except $\lceil u' \rceil.\mathit{QS} = \lceil u \rceil.\mathit{QS} \cup \{q'\}$.

Receiving a reply r from memory A matching request $q' \in \lceil u \rceil.\mathit{QS}$ was sent for a core request q with $\lceil u \rceil.\mathit{state}(q) \in \{\textsc{Trans}\ q', \textsc{Final}\ q'\}$. In the first case r is a page table lookup reply and $\lceil u' \rceil.\mathit{PTL} = \lceil u \rceil.\mathit{PTL} \cup \{r\}$. The next state for q may be $\textsc{Fault}$ in case of translation faults, $\textsc{Trans}\ \bot$ if translation is not finished yet, and FINAL $\perp$ otherwise. In the second case r is the reply for a translated core request. We record $\lceil u' \rceil.RR = \lceil u \rceil.RR \cup \{r\}$ and leave q' status unchanged. In both cases we have $\lceil u' \rceil.QS = \lceil u \rceil.QS \setminus \{q'\}$ and no other changes.

Sending reply r to the core A corresponding request $q \in \lceil u \rceil.QR$ has either status FAULT or FINAL q'. In the first case r is a fault reply computed for q. Otherwise, there is a reply $r' \in \lceil u \rceil.RR$ that matches q' and r is a reversely address-translated version of r'. Moreover, r is not a second-stage translation fault and we have $\lceil u' \rceil.RR = \lceil u \rceil.RR \setminus \{r'\}$. In both cases we also demand $\lceil u' \rceil.QR = \lceil u \rceil.QR \setminus \{q\}$ and $\lceil u' \rceil.state(q) = \perp$.

Secure address translation scheme The behavioral specifications above do not specify when a memory access will fail or succeed as this behavior of the MMU depends on its configuration and the page tables in memory. In order to abstract from the specific page table layout of the architecture and capture the desired memory isolation guarantees, we introduce an uninterpreted notion of *golden* page tables and MMU configurations that implement a given restricted address map for a guest, which, e.g., does not allow accesses to other guests' unshared memory or the locations of the page tables itself.

Assuming that an MMU has such a golden configuration, and has only ever looked up page table entries from a corresponding golden page table, we can further constrain the behavior of the MMU, in particular:

- further page table lookups only target the region of the golden page tables,
- final translations reflect the desired address map,
- faults result only from accesses to unmapped addresses or write permission faults.

In addition, while a request is being translated, we consider it nondeterministic *when* it will reach its final state or result in a fault. Assuming a golden configuration and page tables, we require a progress condition that one of the possible outcomes will occur after a finite amount of MMU lookup steps.

5.3 Memory

In this case study we use a simple coherent, multi-copy atomic shared memory model without architecturally visible side effects from the caches or other parts of the memory subsystem. 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. Memory transitions consist of:

- receiving a request from an MMU or SMMU (in case of device DMA access),
- forwarding a core's MMIO access to a device,
- receiving a device's reply to an MMIO access,
- sending a reply for an earlier memory request.

In the last case, for RAM accesses we require that the usual memory semantics apply, i.e., reads return the contents for requested addresses without changing memory, writes update only the contents for targeted addresses. MMIO replies from devices are just forwarded to the requesting cores. The soundness of the flat memory model is discussed in Sect. 11.

5.4 Devices and System MMUs

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 except for an *active* flag and a number of explicitly defined history variables keeping track of received and sent DMA and MMIO messages. Device transitions consist of:

- receiving and replying to MMIO accesses from memory on behalf of a core,
- sending DMA requests to an SMMU (if *active* holds) and receiving DMA replies,
- sending or receiving external signals associated with that device,
- sending an interrupt associated with that device to the GIC, if *active* holds.

We provide behavioral specifications only, stating sanity checks based on the bookkeeping of messages, e.g., that each sent reply matches a pending MMIO request.

Our synchronous message passing approach fits well with edge-triggered interrupt signaling, but it also covers level-triggered interrupts, depending on the GIC's interpretation of the edge signals.

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 parameterized with different translation schemes,[3] 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.

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

5.5 Generic interrupt controller

The interrupt controller used on our ARM platform (GICv2) contains four different register states:

- the interrupt distributor shared by all cores, and for each core:
  - a physical interrupt interface,
  - a virtual interrupt control interface,
  - a virtual interrupt interface.

In the abstract specification we largely leave the registers underspecified, except for control registers used by the hypervisor.

For modeling the side effects of MMIO accesses to the GIC and interrupt reception, we introduce uninterpreted abstraction 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. The possible GIC actions are:

- receiving an interrupt from a device,
- signaling a physical or virtual interrupt to a core,
- receiving MMIO accesses from the cores to one of the register states,
- replying to an MMIO access.

Side effects of such transitions are expressed rather on the interrupt state than on the register state which can change either nondeterministically or is unaffected by a given transition. However, our behavioral specification constrains the change of registers exposing the current interrupt state to reflect to the prescribed change in the abstract interrupt state. For example, 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. Only for the registers explicitly touched by the hypervisor, e.g., to configure inter-processor interrupts, we model the effect of accessing the registers in detail.

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 IRQ interrupt signals.

6 Hypervisor model

The main functionality of the HASPOC hypervisor [8] is to bring up the platform into a state where different guest systems can run in statically allocated, isolated partitions, such that each guest owns 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 unidirectional shared memory channels between pairs of guests and associated inter-processor notification interrupts that can be requested through hypercalls. There is also a secure boot loader that cryptographically verifies the authenticity of the provided hypervisor and guest images.

For this work we are mainly interested in the platform invariant $Inv : \mathbb{S} \rightarrow \mathbb{B}$ that constrains the abstract states of 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 core and device requests may access only the associated guest's physical memory, including writes to outgoing and reads to incoming IGC channels. In addition, cores may access memory-mapped I/O regions of owned devices and the 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 memory-mapped I/O region. Requests to GIC registers other than the virtual interrupt interface are sent by cores running in hypervisor mode. For reply messages traversing the platform, similar restrictions hold.
- Page tables are stored in hypervisor memory, disjoint from guest memory. The page tables are fixed and, together with the MMU configurations established by the hypervisor initialization phase, implement a secure translation scheme for guests, guaranteeing memory isolation modulo IGC channels.
- 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. Similarly, only interrupts from devices belonging to a given guest may be pending or active at its core's virtual interrupt interfaces.
- Well-formedness invariants are imposed on messages and the abstract state of all SoC components. For instance, for an (S)MMU u we couple the received and sent

memory messages with the translation state according to the behavioral specification. We require, e.g., that $q \in \lceil u \rceil.\mathit{QR}$ holds iff $\lceil u \rceil.\mathit{state}(q) \neq \bot$ or that for replies $r \in \lceil u \rceil.\mathit{RR}$, there exists a request q with $\lceil u \rceil.\mathit{state}(q) = \textsc{Final}\ q'$ such that r matches q' which is an address-translated version of q.
- 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. We identify properties of the abstract component states that must hold in the initial system states $s_0 \in \mathbb{S}_0$ with $s_0(i) \in \mathcal{K}(i).\Sigma_0$ for $i \in I$, e.g., that initially no memory requests or interrupts are pending, devices are inactive, and cores start in EL3 executing the boot loader. Then, we have to prove:

Theorem 1 *All computations $t \vdash s_0 \rightarrow s'$ starting in an initial system state $s_0 \in \mathbb{S}_0$ preserve the platform invariant, i.e., $\mathit{Inv}(s')$ holds.*

The theorem can be proved by induction on n. In particular, the invariant needs to be defined on the state abstractions in such a way that $\mathit{Inv}(s_0)$ holds for all such s_0, requiring certain invariant parts, like the correct configuration of the (S)MMUs, only after they have been established by corresponding phases of the hypervisor initialization process. In the induction step we distinguish all possible component transitions and use their behavioral specifications to show that the invariants are preserved. Decomposition allows some measure of local reasoning as at most two components are changed in one step and others usually retain their invariants as they are unaffected.

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.

In addition to platform initialization, the hypervisor contains handlers for providing virtualized functionality to the guest. In particular, it:

- virtualizes a GIC distributor for each guest, preserving the interrupt isolation invariants,
- handles all other second-stage translation faults and injects them into the core in guest mode,
- receives physical interrupts and registers them as virtual interrupts in the GIC,
- 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 are 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 to cores running in guest mode.

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 [35]. In our case it suffices to design the LTS in such a way that each transition contains at most one step 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 [50].

Formally, we introduce such hypervisor transitions in our system model by adding a new kind of synchronization vector $\textsc{Orcl}\ \omega$, which 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 $\textsc{Orcl}\ \omega \vdash s \rightarrow s'$ iff $(s, s') \in \omega$.

7 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, since the bisimulation property forces the traces in both systems to be identical. Then, we can use our platform to build trustworthy systems where security-critical services are properly isolated from untrusted software. Figure 3 shows the ideal models for two applications that have been successfully implemented on top of the HASPOC hypervisor: a secure network bridge and a secure VPN solution for Android phones. By construction, no direct communication is possible between the untrusted

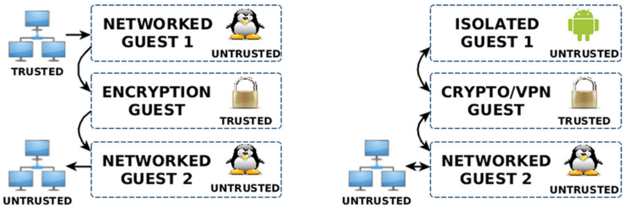

Fig. 3 Use cases: with correctness of the crypto service, guest 2 only receives encrypted traffic from guest 1, even if both guests are compromised. Hence, secrets in guest 1 are secure

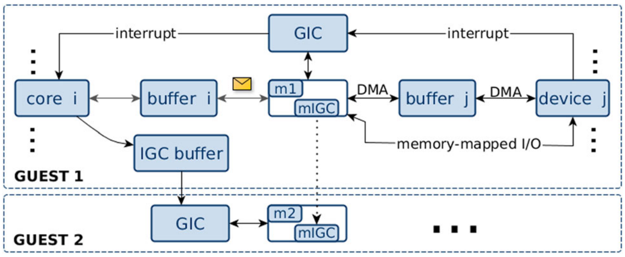

Fig. 4 Ideal model: guest 1 with its share of cores, devices, and memory regions—connected to guest 2 via an inter-guest communication (IGC) interrupt and duplicated but synchronized IGC memory channel

guests and all information leaving guest 1 has to go through an encrypted channel.

In the context of our formal system model, the ideal model is a system where each component is instantiated with one idealized guest SoC. At this level the only interference possible between guests is by message passing through the IGC notification channels, or via external I/O through devices owned by the guest SoCs. 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 a guest modifies one of its the outgoing IGC memory channels.

The idealized guest SoC models are structured as the underlying ARMv8 system model itself, 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:

– Ideal cores execute in guest mode only. At ideal level hypervisor execution is invisible and effects of handlers on the core are modeled explicitly as part of the ideal core semantics. For example, hypercall instructions get special semantics that reflect a complete handler execution and memory access faults that are caught by the hypervisor appear as if they cause a direct jump into the guest's exception handler. 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 ideal memories is restricted to each guest's memory region. (S)MMUs are replaced by simple core and device message 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 message 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, and only interrupts belonging to the guest may ever become pending or active.
– To model IGC notification interrupts between guests, we add a special notification interrupt buffer for each outgoing channel. The buffers are filled by a hypercall instruction with idealized semantics mirroring the behavior of the underlying hypervisor handler. Using a global MSG transition with the receiving guest SoC, an IGC interrupt is injected into the receiver's ideal GIC. This simulates the hypervisor behavior when receiving the physical inter-processor interrupt and registering it as a virtual interrupt.

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, devices in both models receive exactly the same messages and therefore behave identically.

We prove an invariant *IInv* 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* $\bar{s}_0 \in \bar{\mathbb{S}}_0$, *all computations* $\bar{t} \vdash \bar{s}_0 \rightarrow \bar{s}'$ *establish* $\mathit{IInv}(\bar{s}')$.

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, $\mathit{IInv}(\bar{s}_0)$ holds trivially. In the induction step we first show *IInv* 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.

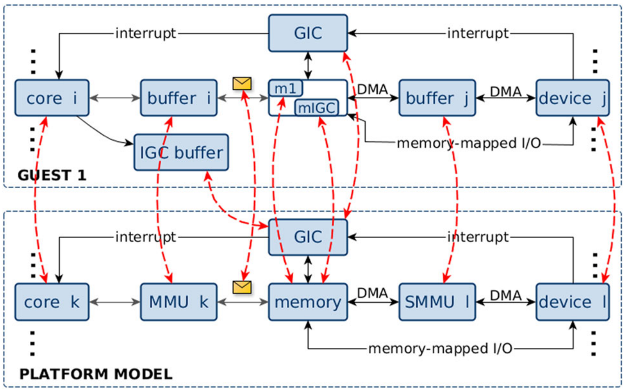

Fig. 5 The bisimulation relation (red dashed arrows) between ideal and platform model is decomposed into component-specific subclauses for each guest (color figure online)

8 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 as illustrated in Fig. 5. 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 and internal states as well as message history variables while the core is running in guest mode. While the hypervisor is running, guest registers are either coupled as above, saved in banked registers, or stored in hypervisor memory, depending on the kind of register and the hypervisor program state.
- 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 normally present in the ideal model if, and only if, they are present as address-translated guest requests/replies in the platform model in corresponding components. 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. In the latter case, write requests in the ideal model are projected to sanitized versions in the platform model, where any updates to protected parts of the GIC register state are removed by the hypervisor. Similarly, all read replies in the platform model appear as filtered versions in the ideal model, containing only information about interrupts belonging to the guest as other protected information will be removed by the hypervisor as well.
- 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. During reconfiguration of the interrupt state by the hypervisor, a more complex coupling is in place, reflecting the current program state of the handler, current hypervisor accesses to the GIC, and the stepping strategy for the bisimulation proof.
- IGC interrupts are active in the ideal IGC notification buffer, while the corresponding SGI between the cores 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 interrupts belonging to other guests.

In general, the coupling of components, memory messages, and interrupts is quite straightforward while the guest is executing, but more complex 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 direct the interested reader to our technical specification [9]. The desired correctness theorem can now be stated in two parts.

Theorem 3 *Given initial states $s_0 \in \mathbb{S}_0$ and $\bar{s}_0 \in \bar{\mathbb{S}}_0$ with $s_0\ \mathcal{R}\ \bar{s}_0$, then (1) for any computation $t \vdash s_0 \rightarrow s'$ of the platform model there exists a corresponding ideal model computation $\bar{t} \vdash \bar{s}_0 \rightarrow \bar{s}'$ such that $s'\ \mathcal{R}\ \bar{s}'$ holds, and (2) for any computation $\bar{t} \vdash \bar{s}_0 \rightarrow \bar{s}'$ of the ideal model there exists a corresponding platform model computation $t \vdash s_0 \rightarrow s'$ with $s'\ \mathcal{R}\ \bar{s}'$.*

Theorem 3 is proved by induction on n for the two directions separately. The base case is trivial. In the induction step we 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 bisim-

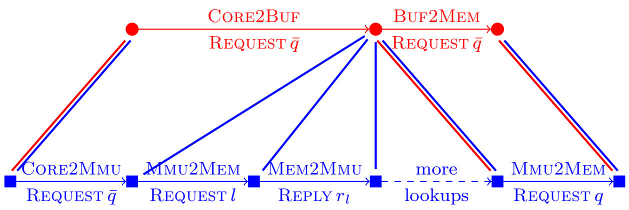

Fig. 6 Bisimulation of a successful MMU translation of core request $\bar{q}$ to q, matching platform steps (blue, below) with ideal steps (red, above). The dashed arrow subsumes a finite number of page table lookup steps (MMU2MEM, MEM2MMU). Blue lines between platform and ideal states show the bisimulation relation $\mathcal{R}$. Red lines highlight the stepping strategy when simulating the ideal computation: All address translation steps are executed for the ideal CORE2BUF step; all intermediate states are coupled with the resulting ideal state (color figure online)

ulation relation again, preserving the properties sketched above.

Existence argument While showing the existence of simulating steps, we apply our enabling specifications, deriving the sufficient precondition E from relation $\mathcal{R}$ and the invariants. Thus, we can conclude that the desired corresponding transition is indeed enabled.

However, recall that we do not provide enabling specifications for all transitions. For instance, given a platform core transition from σ to σ', sending a memory request m, it depends on the uninterpreted internal core state, if this step is indeed possible in a coupled ideal core state $\bar{\sigma}$. To solve the issue, we require a *bisimulation obligation* in the shape of (2) that links identical transitions on the ideal and platform model wrt. an abstract core coupling relation $\mathcal{R}_{\text{core}}$ that is implied by the bisimulation relation $\mathcal{R}$ on the system states. In the direction from platform to ideal model (and vice versa), we demand the following property:

$$\forall \bar{\sigma}, \sigma, m, \sigma'.\ \lceil \sigma \rceil\ \mathcal{R}_{\text{core}}\ \lceil \bar{\sigma} \rceil \wedge \text{snd}(\sigma, m, \sigma) \Rightarrow \\ \exists \bar{\sigma}'.\ \text{snd}(\bar{\sigma}, m, \bar{\sigma}') \wedge \lceil \sigma' \rceil\ \mathcal{R}_{\text{core}}\ \lceil \bar{\sigma}' \rceil\ .$$

Thus, we obtain the existence of the corresponding ideal core step, as well as the required coupling directly from a proof obligation that has to be discharged for given platform and ideal core instantiations. As $\mathcal{R}_{\text{core}}$ requires the registers as well as the internal state of the cores to be equal, and since we can define the ideal core model freely, this should be a straightforward exercise. However, the proof requires to show that the core architecture does not leak information from higher exception level registers into guest registers [31, 47].

We demand a similar bisimulation obligation for guest accesses to the virtual GIC interfaces, as their semantics is widely underspecified (cf. Sect. 5.5). Hence, also any instantiation of the ideal GIC model needs to match the semantics of the platform model tightly.

Stepping Strategy and Proving $\mathcal{R}$ Whenever the hypervisor handlers are not involved, steps of both models are mostly mapped in a one-to-one fashion. Nevertheless, the translation steps of the (S)MMUs are invisible in the ideal model. When simulating an ideal core or device sending a request to its memory interface buffer, we step the (S)MMU until either a fault occurs or the translation is successful, using the (S)MMU progress condition. Figure 6 depicts the process for sending a request from core to memory. Similarly, we step hypervisor handlers to completion in the platform model when simulating the corresponding ideal transitions.

We add an induction hypothesis to the simulation of the ideal by the platform model encoding 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 distinguish if the simulated step is corresponding to zero, one, or more steps of the simulating model. In the latter case we identify one step in the simulating sequence where we step both models simultaneously. For preceding or subsequent steps the simulated model stutters and we prove that the bisimulation relation with the pre- or poststate of the simulated transition is preserved. Recall, that such stuttering only occurs during handler execution and address translation. In all cases we profit from the compositional approach: As at most two components change in one model, clauses of $\mathcal{R}$ relating other components are usually preserved trivially.

For clauses related to the implementation of handlers, however, it may happen that other components need to be taken into account since here the coupling of ideal and platform model components may be temporarily out of sync and depend on the state of other components modified by the hypervisor. For instance, the state of interrupts in the ideal model depends on the progress of the hypervisor interrupt handler sending messages through the memory system to the GIC. Interrupts are injected in the ideal model (virtualized) core interrupt interfaces as soon as the GIC answers an interrupt acknowledgment request by the hypervisor, as this is the step at which the hypervisor commits to injecting the interrupt into the virtual interface.

To establish the relevant clauses of $\mathcal{R}$ for a given component, we apply the system invariants and the behavioral specification of corresponding steps. For instance, the final pair of steps in Fig. 6 send a (translated) memory request $\bar{q}$ (q) from the message buffer $\bar{u}$ (MMU u) to memory $\bar{m}$ (m). From the MMU semantics we know $\lceil u \rceil(\bar{q}) = \text{FINAL}\ \bot$ and that q is an address-translated version of $\bar{q}$. By *Inv* we have that the MMU is configured according to the secure address translation scheme that is also used for the coupling of guest messages in $\mathcal{R}$; thus, if $\bar{q}$ targets an intermediate physical address in guest memory, then q targets the corresponding address in physical memory. The behavioral specifications require updates $\lceil u' \rceil.QS = \lceil u \rceil.QS \cup \{q\}$ and $\lceil m' \rceil.QR = \lceil m \rceil.QR \cup \{q\}$ for the message bookkeeping components of MMU and memory, and similarly $\lceil \bar{u}' \rceil.QS = \lceil \bar{u} \rceil.QS \cup \{\bar{q}\}$ and $\lceil \bar{m}' \rceil.QR = \lceil \bar{m} \rceil.QR \cup \{\bar{q}\}$ in the ideal model of the corresponding guest. Components of other guests are not affected, since the MMU is dedicated to a single core (and thus a single guest) and the memory contents are unchanged. The clause in $\mathcal{R}$ that couples the sent MMU requests with the sent core requests in the ideal message buffer is preserved because the same requests are added modulo the secure address translation scheme. We argue similarly for the requests now pending in $\lceil m' \rceil.QR$ and $\lceil \bar{m}' \rceil.QR$.

For steps that update memory contents, the same strategy is used, stating first that the same address is updated in guest memory (modulo address translation) and using the memory update semantics to show that the same effects occur. For writes to shared memory we also need to take into account the synchronization between IGC memory channels in the ideal world. Before the synchronization is performed only the sending guest's IGC memory is coupled with the platform memory contents. After the synchronization the coupling of memory contents holds also for the receiving guest.

9 Application: transfer of confidentiality

Having proved the bisimulation theorem, we can apply it to transfer safety and information flow properties of the ideal model to the platform model. A caveat is that we can only transfer properties that can be expressed in terms of the abstract states, as the bisimulation relation is agnostic to any underlying instantiations. On the other hand, this approach has the benefit that property transfer only has to be proved once for a given property, and that it applies to all valid instantiations of the models. To give an example, we sketch how confidentiality of secrets within one guest of the model can be established on the platform model by proving it on the ideal model and using the bisimulation to transfer it.

Assume victim V is a guest of the system and all other guests are controlled by an attacker A (cf. Fig. 3, where V is the crypto process). There is a secret stored in V that shall not leak to A. The confidentiality of the secret can be expressed as a noninterference property wrt. the possible observations of A in the system [20,44]. At a high level the observations in the ideal model are determined by the sequence of A's input messages. At a lower level we define observation functions on the abstraction of state components in the system, to clarify which parts of the abstract states are indeed observable for A. Naturally, in the ideal model all components in all guests except V are observable.

We now consider transition sequences that have the same transitions for A (including the same input messages) but may differ for other actions of V. If we execute such schedules in ideal system states that are indistinguishable for A, i.e., they have the same observations, then we can apply the *completeness* property of our abstract specifications to show that the resulting states are still indistinguishable. In particular, since the observations of A cover the complete abstract state of its components, indistinguishability of two component states $\bar{\sigma}_1, \bar{\sigma}_2$ means $\lceil \bar{\sigma}_1 \rceil = \lceil \bar{\sigma}_2 \rceil$. Then, the completeness property yields that for each transition on state $\bar{\sigma}_1$, the same transition is enabled on $\bar{\sigma}_2$, resulting in the same abstract states and thus the same observations.

In order to establish noninterference on the whole ideal model, it remains to show that the sequences of inputs sent by V to A are independent of the secret. If there is no IGC communication channel from V to A, there are no inputs to A except the external ones. Assuming these are independent of V's external outputs, there is nothing left to show. In what follows we focus on the more interesting case, where there is an IGC channel from V to A. Additional inputs to A are V's writes into the IGC memory region and IGC notification interrupts. If these events are secret-independent, e.g., if V never performs secret-dependent accesses to IGC memory or requests of IGC notification interrupts, the same sequence of inputs to A is indeed generated for all possible secret values. Note that, this property can usually not be proved without considering a specific instantiation of V that includes its program code.

The ideal noninterference property now states that for initial indistinguishable states $\bar{s}_1, \bar{s}_2$ and schedule $\bar{t}_1$ such that $\bar{t}_1 \vdash \bar{s}_1 \rightarrow \bar{s}'_1$, there exists a schedule $\bar{t}_2$ with the same sequence of transitions and inputs for A such that $\bar{t}_2 \vdash \bar{s}_2 \rightarrow \bar{s}'_2$ and $\bar{s}'_2$ is indistinguishable from $\bar{s}'_1$.

Next, we define the observations of A on the platform model. The corresponding input events for A are external inputs, writes by V to the shared IGC memory, and the registration of an IGC inter-processor interrupt from V to A in the GIC distributor. As the memory and GIC are shared between all guests, the definition of the observation functions is more tricky. We omit the states of all cores, (S)MMUs, devices, and GIC core interfaces belonging to V from the observations of A. Similarly, we remove all memory contents and messages belonging to the unshared memory of V. For the GIC distributor we filter out all information belonging to interrupts of V from the interrupt and register states, except for IGC notification interrupts. We similarly filter out this information from corresponding data structures in hypervisor memory, which is otherwise observable by A as it influences the execution of the handlers that A invokes.

The observations of A are additionally restricted by the bisimulation, as not all abstract components in the platform model are covered by relation $\mathcal{R}$. For instance, the intermediate states of (S)MMU translation are not linked to any ideal component. Similarly, the stepping strategy for hypervisor handlers hides intermediate hypervisor program states. Intuitively, these states do not reveal any secrets and the attacker has no way to directly observe them; therefore, we only consider platform computations that exhibit the block-wise stepping strategy of the bisimulation proof, i.e., none of these intermediate states are ever exposed.[4]

We then need to show that indistinguishable platform states are also indistinguishable in the ideal model when coupled via $\mathcal{R}$ and vice versa. Since the abstract ideal and platform component models match so tightly, and because we hide intermediate translation and hypervisor states, this is straightforward.

Finally, we prove noninterference as follows. Assume ideal and platform initial states $\bar{s}_1$, $\bar{s}_2$, s_1, and s_2 such that $s_1 \; \mathcal{R} \; \bar{s}_1$ and $s_2 \; \mathcal{R} \; \bar{s}_2$. Furthermore, consider a schedule t_1 resulting in s_1', i.e., $t_1 \vdash s_1 \rightarrow s_1'$, and let s_1 and s_2 be indistinguishable for A. As discussed above, this means that also $\bar{s}_1$ and $\bar{s}_2$ are indistinguishable. By Theorem 3 we obtain a corresponding schedule $\bar{t}_1$, and $\bar{t}_1 \vdash \bar{s}_1 \rightarrow \bar{s}_1'$ such that $s_1' \; \mathcal{R} \; \bar{s}_1'$. By the noninterference property of the ideal model, we get a schedule $\bar{t}_2$ with the same transitions and inputs for A as in $\bar{t}_1$, and $\bar{t}_2 \vdash \bar{s}_2 \rightarrow \bar{s}_2'$ such that $\bar{s}_2'$ is indistinguishable from $\bar{s}_1'$. Applying Theorem 3 again in the other direction, we deduce noninterference on the platform model, i.e., there exists a schedule t_2 with the same transitions and inputs for A as t_1 and a platform state s_2', such that $t_2 \vdash s_2 \rightarrow s_2'$ and s_2' is indistinguishable from s_1' for A.

The bisimulation result also shows that the only channels available for a malicious guest system to influence the hypervisor or another guest system is to use one of the authorized guest-to-guest communication channels or one of the hypercalls. In particular, any attempt of attacker A to access memory outside its assigned memory range will lead to a fault in both the ideal and the platform model.

10 Implementation

We have executed our case study in the theorem prover HOL4 and proved information flow security for the guest execution [9]. In particular, the induction step for the bisimulation property (Theorem 3) comprises 13 hypervisor transitions and 42 transitions that at least partly concern guest execution in the platform and ideal model combined. For the latter number of guest steps, we have shown formally that they preserve all 24 clauses of the bisimulation relation. The ideal invariant (Theorem 2) is verified completely, while the platform invariant (Theorem 1) is only proven on paper so far [9].

Table 1 HOL4 lines of code for basic parts (general data types etc.), the ideal model, the platform model, the hypervisor, as well as the bisimulation relation and proof

	Basic	Platf.	Hyp.	Ideal	Bisim.	Total
Model	154	1756	1931	1124	669	5634
Invariant	–	613	–	212	–	825
Tools	525	–	–	198	675	1398
Proofs	773	2197	1633	1922	15,316	21,841
Total	1452	4566	3564	3456	16,660	29,698

Table 1 provides an overview on the size of the different parts of our development. We consider "basic" parts and the developed proof tools as reusable for similar endeavors. The case study so far took about 15 person months with low to intermediate level of a priori HOL4-expertise. As can be seen from the numbers, the modeling effort was moderate considering the complexity of the underlying platform and the proofs make up by far the largest part of the development, as one would expect for an interactive proof system. The numbers also hint at the increased complexity of the platform invariant (and presumably the effort required to formalize its proof) compared to the ideal model invariant.

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 would focus on one transition step at a 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 poststates, 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. As future work, we plan to extend the machinery by the automatic identification and verification of required pre- and postconditions according to our abstract transition specifications.

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 mod-

[4] An alternative approach could set the observations in intermediate states to the observations at the beginning or the end of a block, depending on the stepping strategy.

els that provides a lot of these properties by construction for a given instantiation and avoids the pitfalls discovered in our ad hoc definition.

While conducting formal proofs can be tedious at times, we experienced that the decomposed modeling approach indeed helps speeding up the verification engineering process. On the one hand, as system transitions in the decomposed model only modify one or two components at a time, proof goals on all other components could mostly be discharged automatically using resolution solvers. On the other hand, the proofs seem to be quite robust against local modifications of the models (which are inevitable when formalizing models of substantial size), i.e., verification results on unrelated parts of the system state were usually unaffected by the changes. We see these observations as further evidence to our premise that decomposition is an indispensable tool in the verification of complex systems.

11 Discussion

The methodology described in this paper combines abstraction and compositional reasoning to prove reusable top-level theorems of information flow security. As shown above, it allows to verify properties like integrity and confidentiality for the design of complex execution platforms with moderate effort. Nevertheless, there are some assumptions underlying our case study and limitations inherent to the approach that we want to discuss below.

Memory model In our case study we assumed a flat, coherent, multi-copy atomic memory model. As of late, the ARMv8 processors are required to implement a multi-copy atomic memory semantics. Hence, all of the weakly consistent memory behavior results from out-of-order execution in the cores [40]. Therefore, soundness of the memory model presupposes only that neither the hypervisor nor the guests break memory coherency. This 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 [23]. Similar effects can be created using Rowhammer attacks [48]. We did not consider these attack vectors when defining our platform model, a methodology to repair proofs of integrity in the presence of such storage side channels is presented in [36]. In brief, the idea is to implement countermeasures and enrich the platform invariant, guaranteeing that the original platform semantics without storage channels is preserved.

Hypervisor model Our formalization of the hypervisor handlers needs to correctly capture the actual implementation of the hypervisor. In particular, as the hypervisor is a concurrent piece of software, each step in the hypervisor LTS needs to be implemented by an atomic block of code that is implementing a locking policy to synchronize on shared hypervisor data structures and the GIC distributor. Validating these assumptions requires at least a method for verifying lock implementations in weakly consistent memory semantics [2,53], an order reduction argument [35], and a binary code verification approach [24]. Consisting of some 8000 lines of source code, formal verification of the hypervisor implementation is feasible, albeit out of scope for this work.

Timing channels The platform model presented above does not model time explicitly; hence, the hypervisor design is potentially susceptible to timing side-channel attacks [19]. However, the attack surface is reduced as cores and accompanying L1 caches are not shared by different guests. Nevertheless, the cores are shared with the hypervisor and its timing behavior when accessing shared hypervisor data structures could in principle leak information about hypervisor thread activity on behalf of other guests. Attacks based on speculative execution seem a slightly more credible thread. While MMU-circumventing attacks like Foreshadow-NG [55] do not seem to be possible on ARM processors, Spectre-type attacks [33] on hypervisor handlers may enable an attacker to read hypervisor data structures and leak information about the behavior of other guests. The biggest thread to information flow security is possibly the last-level cache that is shared between all cores [29]. Timing attacks in this concurrent scenario can be prevented by a cache-coloring scheme [30]; however, in the presence of communication channels, the performance overhead is likely to be high, as each IGC channel would need a separate cache color to isolate shared from private memory access behavior.

Message definitions As discussed earlier, the bisimulation theorem proved in this work can only transfer properties that are expressible on the abstract component models. The same issue arises when considering the definition of messages $\mathbb{M}$ of the platform. If the definition is precise, i.e., if it does not contain uninterpreted placeholder variables that can be refined by an instantiation later, then the information that can be exchanged by components is fixed. For instance, in our case study, the memory replies $\mathbb{R}$ have a fixed definition and they only contain the original request as well as a return value for reads or fault information. Thus, in hindsight we cannot easily add additional information, e.g., about cache hits or misses, to the memory reply messages without adapting the overall proof.

Leaving parts of messages underspecified, as we do for requests $\mathbb{Q}$, allows to add information during instantiation of the platform model, but this comes at a cost when verifying transitions that create such messages. For instance, when proving the bisimulation for sending transitions of ideal and platform cores, we need a bisimulation obligation to ensure that the same message is sent by coupled core components.

Moreover, instantiations of components receiving such messages may only use the additional information in ways that are consistent with their behavioral specification. For memories in particular, this means that the additional message information in requests must not affect the memory contents when executing read or write requests.

Modeling shared caches As argued above, timing channels pose a security thread to the confidentiality of secret guest data, mainly due to the shared last-level cache. While a precise model of the platform's timing behavior seems out of reach, at least the cache state may be exposed in order to be able to reason about cache-based channels. It should be straightforward to add an explicit cache model to the abstract states and transition specifications of ideal and platform memories. If the hypervisor implemented a cache-coloring scheme, the platform invariants have to be adapted to constrain the mapping of guest memory addresses to cache lines. The bisimulation relation would map the corresponding parts of the cache state to partial cache states in each guest, in a similar fashion as we couple the shared GIC distributor state. In its current shape, the hypervisor leaves the handling of timing channels to the guests. Thus, if caches are modeled, the ideal model would need to contain an additional communication channel between all guests, synchronizing the cache state for each memory access in the platform.

Complications arise if cache state and semantics are left uninterpreted at the abstract level, i.e., to be defined precisely during instantiation. In principle, this is desirable, as it allows the overall bisimulation to cover any cache implementation. In practice, however, this approach would require bisimulation obligations to couple the uninterpreted cache states in the ideal and platform model. In any case, the abstract models would have to take the presence or absence of countermeasures employed by the hypervisor into account. Defining an uninterpreted model of shared caches, that can be successfully instantiated, seems like a hard problem.

An alternative, potentially more practical approach is to leave the caches out of the abstract models and handle cache-based information flow at the instantiated platform level. Similar to the handling of cache-based storage channels [36], the platform invariant could be enriched to cover countermeasures that prevent the cache state to leak secret information of one guest to another, e.g., cache partitioning by the hypervisor, or secret-independent memory accesses by a guest. Then, having confidentiality on the platform model, modulo the cache state, the observations of the attacker can be extended to include the cache state as well. The correctness of the countermeasures then ensures that also the cache states of an attacker are indistinguishable for different values of a victim's secret.

Defining abstractions The definitions of the abstract states and transition specifications for the platform components are the cornerstone of this work. Throughout the proof process, we have identified specifications that are sufficiently strong to allow the bisimulation to be proved. The bisimulation proof guarantees that all detailed component instantiations, satisfying the abstract specifications, exhibit the same secure information flow as in the ideal model.

Naturally, it is easy to make too strong assumptions here; hence, it is crucial to discharge the proof obligations on detailed component models. If such a proof fails, the abstract state and abstraction function, transition specifications, or the coupling relation for that component need to be adapted. Here, the decomposed verification approach reduces the cost of re-verification, too, as hopefully only local changes are required.

Trustworthy hardware models When instantiating components and proving that abstractions correctly model their behavior, there is a shift in the assumptions of our bisimulation theorem. Instead of the relying on the correctness of the abstractions, the soundness of the theorem now relies on the assumption that the detailed component models describe the behavior of the actual hardware correctly. This is a common problem for all formal treatments of computer systems and not specific to our approach. In the absence of a machine-readable hardware specification provided by the vendors [41], trust in the models can be increased by simulating them against the actual hardware [3] or other extensively tested hardware emulators [10,11]. Recently, there have also been efforts to synthesize instruction semantics using machine learning [28]. While these approaches usually target processor semantics, their application to system components like (S)MMUs, devices, or the interrupt controller, are an open problem.

A more philosophical question is why other people should trust a formal development that is as large as ours. Leaving aside the correctness of HOL4 (which we take for granted), a critical reader of our work, e.g., a certification agent, would have to assess all our definitions and formulations of theorems in order to judge if they correctly capture the problem. To assist such a process, we provide extensive technical documentation and guidance describing the formal artifacts and their underlying intentions [9].

Nevertheless, these are in nature just informal assurances. We believe that a more structured approach to the creation of platform specifications and top-level proofs is a way to greatly increase trust in formal verification. Having a formal framework that enshrines the methodology outlined in this paper ensures that models, specifications, and theorems are presented in a uniform style and proofs may be increasingly automated. Moreover, it allows to apply sanity checks on the given definitions and the integration of domain-specific languages to either define models in a more readable way or import existing ones automatically.

12 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 by 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). The discussed case study provided valuable insights toward that goal. Further directions of work include discharging the proof obligations posed by our abstract transition specifications for the existing detailed component models of ARMv8 core and MMU, transferring information flow properties along the lines of Sect. 9, and verifying countermeasures against cache-based information leakage.

Acknowledgements We kindly thank Thomas Tuerk for his help in improving our HOL4 definitions. This work was supported by the PROSPER Project funded by the Swedish Foundation for Strategic Research (SSF), 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. Abadi, M., Lamport, L.: An old-fashioned recipe for real time. ACM Trans. Program. Lang. Syst. **16**(5), 1543–1571 (1994). https://doi.org/10.1145/186025.186058
2. Alglave, J., Kroening, D., Nimal, V., Tautschnig, M.: Software verification for weak memory via program transformation. In: European Symposium on Programming, pp. 512–532. Springer (2013)
3. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. (TOPLAS) **36**(2), 7 (2014)
4. Alkassar, E., Hillebrand, M.A., Paul, W.J., Petrova, E.: Automated verification of a small hypervisor. In: Proceedings of VSTTE, LNCS, vol. 6217, pp. 40–54. Springer (2010)
5. Alur, R., Dang, T., Esposito, J., Fierro, R., Hur, Y., Ivančić, F., Kumar, V., Lee, I., Mishra, P., Pappas, G., Sokolsky, O.: Hierarchical hybrid modeling of embedded systems. In: Embedded Software (EMSOFT), pp. 14–31. Springer (2001). https://doi.org/10.1007/3-540-45449-7_2
6. Barthe, G., Betarte, G., Campo, J.D., Luna, C.: Formally verifying isolation and availability in an idealized model of virtualization. In: Formal Methods, pp. 231–245 (2011)
7. Barthe, G., Betarte, G., Campo, J.D., Luna, C.: Cache-leakage resilient os isolation in an idealized model of virtualization. In: Proceedings of CSF'12, pp. 186–197. IEEE (2012). https://doi.org/10.1109/CSF.2012.17
8. Baumann, C., Näslund, M., Gehrmann, C., Schwarz, O., Thorsen, H.: A high assurance virtualization platform for ARMv8. In: European Conference on Networks and Communications (EuCNC), pp. 210–214 (2016)
9. Baumann, C., Schwarz, O., Dam, M.: GitHub repository of formal artifacts and technical documentation. https://github.com/rauhbein/haspocproofs. Accessed 23 May 2019
10. Bellard, F.: QEMU, a fast and portable dynamic translator. In: USENIX Annual Technical Conference, FREENIX Track, vol. 41, p. 46 (2005)
11. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M.D., Wood, D.A.: The gem5 simulator. SIGARCH Comput. Archit. News **39**(2), 1–7 (2011). https://doi.org/10.1145/2024716.2024718
12. Bolignano, P., Jensen, T., Siles, V.: Modeling and abstraction of memory management in a hypervisor. In: FASE/ETAPS, pp. 214–230. Springer (2016)
13. Chen, H., Wu, X.N., Shao, Z., Lockerman, J., Gu, R.: Toward compositional verification of interruptible OS kernels and device drivers. In: Proceedings of Programming Language Design and Implementation, PLDI'16, pp. 431–447. ACM (2016). https://doi.org/10.1145/2908080.2908101
14. Cimatti, A., Tonetta, S.: Contracts-refinement proof system for component-based embedded systems. Sci. Comput. Program. **97**, 333–348 (2015)
15. Dam, M., Guanciale, R., Khakpour, N., Nemati, H., Schwarz, O.: Formal verification of information flow security for a simple ARM-based separation kernel. In: Proceedings of Computer and Communications Security, CCS'13, pp. 223–234. ACM (2013)
16. Feiertag, R.J., Neumann, P.G.: The foundations of a provably secure operating system (PSOS). In: National Computer Conference, pp. 329–334. AFIPS Press (1979)
17. Fox, A.C.J.: Improved tool support for machine-code decompilation in HOL4. In: Interactive Theorem Proving (ITP), pp. 187–202 (2015)
18. Gajski, D.D., Vahid, F.: Specification and design of embedded hardware–software systems. IEEE Des. Test Comput. **12**(1), 53–67 (1995). https://doi.org/10.1109/54.350695
19. Ge, Q., Yarom, Y., Cock, D., Heiser, G.: A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. J. Cryptogr. Eng. **8**(1), 1–27 (2018)
20. Goguen, J.A., Meseguer, J.: Security policies and security models. In: Security and Privacy, 1982 IEEE Symposium on, pp. 11–11. IEEE (1982)
21. Gu, L., Vaynberg, A., Ford, B., Shao, Z., Costanzo, D.: CertiKOS: a certified kernel for secure cloud computing. In: Proceedings of the Second Asia-Pacific Workshop on Systems, APSys'11, p. 3. ACM (2011)
22. Gu, R., Shao, Z., Chen, H., Wu, X., Kim, J., Sjöberg, V., Costanzo, D.: CertiKOS: An extensible architecture for building certified

concurrent OS kernels. In: Operating Systems Design and Implementation, pp. 653–669. USENIX Association (2016)
23. Guanciale, R., Nemati, H., Baumann, C., Dam, M.: Cache storage channels: alias-driven attacks and verified countermeasures. In: Security and Privacy, pp. 38–55 (2016). https://doi.org/10.1109/SP.2016.11
24. Guanciale, R., Nemati, H., Dam, M., Baumann, C.: Provably secure memory isolation for linux on ARM. J. Comput. Secur. **24**(6), 793–837 (2016). https://doi.org/10.3233/JCS-160558
25. HASPOC Project. http://haspoc.sics.se/. Accessed 23 May 2019
26. Hawblitzel, C., Howell, J., Lorch, J.R., Narayan, A., Parno, B., Zhang, D., Zill, B.: Ironclad apps: end-to-end security via automated full-system verification. In: Operating Systems Design and Implementation, pp. 165–181. USENIX Association (2014)
27. He, N., Kroening, D., Wahl, T., Lau, K.K., Taweel, F., Tran, C., Rümmer, P., Sharma, S.: Component-based design and verification in X-MAN. In: Proceedings of Embedded Real Time Software and Systems (2012)
28. Heule, S., Schkufza, E., Sharma, R., Aiken, A.: Stratified synthesis: automatically learning the x86-64 instruction set. In: ACM SIGPLAN Notices, vol. 51, pp. 237–250. ACM (2016)
29. Inci, M.S., Gulmezoglu, B., Irazoqui, G., Eisenbarth, T., Sunar, B.: Cache attacks enable bulk key recovery on the cloud. In: International Conference on Cryptographic Hardware and Embedded Systems, pp. 368–388. Springer (2016)
30. Kessler, R.E., Hill, M.D.: Page placement algorithms for large real-indexed caches. ACM Trans. Comput. Syst. (TOCS) **10**(4), 338–359 (1992)
31. Khakpour, N., Schwarz, O., Dam, M.: Machine assisted proof of ARMv7 instruction level isolation properties. In: Certified Programs and Proofs, pp. 276–291. Springer (2013)
32. Klein, G., Andronick, J., Elphinstone, K., Murray, T.C., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an OS microkernel. ACM Trans. Comput. Syst. **32**(1), 2 (2014). https://doi.org/10.1145/2560537
33. Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: exploiting speculative execution. arXiv preprint arXiv:1801.01203 (2018)
34. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y.: Meltdown: reading kernel memory from user space. In: 27th USENIX Security Symposium (USENIX Security 18), pp. 973–990 (2018)
35. Lipton, R.J.: Reduction: a method of proving properties of parallel programs. Commun. ACM **18**(12), 717–721 (1975)
36. Nemati, H., Baumann, C., Guanciale, R., Dam, M.: Formal verification of integrity-preserving countermeasures against cache storage side-channels. In: International Conference on Principles of Security and Trust (POST 2018), pp. 109–133. Springer (2018)
37. Nemati, H., Guanciale, R., Dam, M.: Trustworthy virtualization of the ARMv7 memory subsystem. In: SOFSEM, pp. 578–589. Springer (2015). https://doi.org/10.1007/978-3-662-46078-8_48
38. Nohl, K., Lell, J.: Badusb–On Accessories that Turn Evil. Black Hat USA, Las Vegas (2014)
39. Paul, W.J., Schmaltz, S., Shadrin, A.: Completing the automated verification of a small hypervisor—assembler code verification. In: SEFM, Lecture Notes in Computer Science, vol. 7504, pp. 188–202. Springer (2012)
40. Pulte, C., Flur, S., Deacon, W., French, J., Sarkar, S., Sewell, P.: Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. Proc. ACM Program. Lang. **2**(POPL), 19 (2017)
41. Reid, A.: Trustworthy specifications of ARMv8-A and v8-M system level architecture. In: Proceedings of Formal Methods in Computer-Aided Design (FMCAD), pp. 161–168. IEEE (2016)
42. RISC-V Foundation: RISC-V—The Free and Open RISC Instruction Set Architecture. https://riscv.org/. Accessed 23 May 2019
43. Rowson, J.A., Sangiovanni-Vincentelli, A.: Interface-based design. In: Proceedings of the 34th Annual Design Automation Conference, DAC'97, pp. 178–183. ACM (1997). https://doi.org/10.1145/266021.266060
44. Rushby, J.: Noninterference, Transitivity, and Channel-Control Security Policies. SRI International, Computer Science Laboratory, Menlo Park (1992)
45. Sang, F.L., Lacombe, E., Nicomette, V., Deswarte, Y.: Exploiting an I/OMMU vulnerability. In: 2010 5th International Conference on Malicious and Unwanted Software pp. 7–14. IEEE (2010)
46. Schwarz, O., Dam, M.: Formal verification of secure user mode device execution with DMA. In: Hardware and Software: Verification and Testing (HVC), No. 8855 in Lecture Notes in Computer Science, pp. 236–251 (2014). https://doi.org/10.1007/978-3-319-13338-6_18
47. Schwarz, O., Dam, M.: Automatic derivation of platform noninterference properties. In: International Conference on Software Engineering and Formal Methods, pp. 27–44. Springer, Cham (2016)
48. Seaborn, M., Dullien, T.: Exploiting the DRAM rowhammer bug to gain kernel privileges. In: Black Hat 15 (2015). https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html. Accessed 23 May 2019
49. Sewell, P., Vitek, J.: Secure composition of insecure components. In: Computer Security Foundations, CSFW'99, p. 136. IEEE Computer Society (1999)
50. Sewell, T.A.L., Myreen, M.O., Klein, G.: Translation validation for a verified OS kernel. In: Programming Language Design and Implementation, pp. 471–482 (2013). https://doi.org/10.1145/2491956.2462183
51. Stewin, P., Bystrov, I.: Understanding DMA malware. In: Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), pp. 21–41 (2012). https://doi.org/10.1007/978-3-642-37300-8_2
52. Syeda, H., Klein, G.: Reasoning about translation lookaside buffers. In: LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, EPiC Series in Computing, vol. 46, pp. 490–508. EasyChair (2017)
53. Vafeiadis, V.: Program verification under weak memory consistency using separation logic. In: International Conference on Computer Aided Verification, pp. 30–46. Springer (2017)
54. Vasudevan, A., Chaki, S., Maniatis, P., Jia, L., Datta, A.: überSpark: enforcing verifiable object abstractions for automated compositional security analysis of a hypervisor. In: 25th USENIX Security Symposium (USENIX Security 16). USENIX Association (2016)
55. Weisse, O., Van Bulck, J., Minkin, M., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Strackx, R., Wenisch, T.F., Yarom, Y.: Foreshadow-NG: breaking the virtual memory abstraction with transient out-of-order execution. Technical Report (2018)
56. Wilhelm, R., Grund, D., Reineke, J., Schlickling, M., Pister, M., Ferdinand, C.: Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **28**(7), 966 (2009)
57. Wojtczuk, R.: Subverting the Xen hypervisor. Black Hat USA, Las Vegas (2008)
58. Xie, F., Yang, G., Song, X.: Component-based hardware/software co-verification for building trustworthy embedded systems. J. Syst. Softw. **80**(5), 643–654 (2007). https://doi.org/10.1016/j.jss.2006.08.015

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.