<http://www.diva-portal.org>

Postprint

This is the accepted version of a paper published in *Journal of Computer Security*. This paper has been peer-reviewed but does not include the final publisher proof-corrections or journal pagination.

Citation for the original published paper (version of record):

Guanciale, R., Nemati, H., Dam, M., Baumann, C. (2016)

Provably secure memory isolation for Linux on ARM.

Journal of Computer Security, 24(6): 793-837

<https://doi.org/10.3233/JCS-160558>

Access to the published version may require subscription.

N.B. When citing this work, cite the original published paper.

Permanent link to this version:

<http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-197900>

Provably Secure Memory Isolation for Linux on ARM

Roberto Guanciale, Hamed Nemati, Mads Dam, Christoph Baumann
Department of Computer Science,
KTH Royal Institute of Technology, Stockholm, Sweden
E-mail: {robertog, hnnemati, mfd, cbaumann}@kth.se

Abstract

The isolation of security critical components from an untrusted OS allows to both protect applications and to harden the OS itself. Virtualization of the memory subsystem is a key component to provide such isolation. We present the design, implementation and verification of a memory virtualization platform for ARMv7-A processors. The design is based on direct paging, an MMU virtualization mechanism previously introduced by Xen. It is shown that this mechanism can be implemented using a compact design, suitable for formal verification down to a low level of abstraction, without penalizing system performance. The verification is performed using the HOL4 theorem prover and uses a detailed model of the processor. We prove memory isolation along with information flow security for an abstract top-level model of the virtualization mechanism. The abstract model is refined down to a transition system closely resembling a C implementation. Additionally, it is demonstrated how the gap between the low-level abstraction and the binary level can be filled, using tools that check Hoare contracts. The virtualization mechanism is demonstrated on real hardware via a hypervisor hosting Linux and supporting a tamper-proof run-time monitor that provably prevents code injection in the Linux guest.

1 Introduction

A basic security requirement for systems that allow software to execute at different levels of security is memory isolation: The ability to store a secret or to enforce data integrity within a designated part of memory and prevent the contents of this memory to be affected by, or leak to, parts of the system that are not authorised to access it. Without the usage of special hardware, trustworthy memory isolation is dependent on the OS kernel being correctly implemented. However, given the size and complexity of modern OSs, the vision of comprehensive and formal verification of commodity OSs is as distant as ever.

An alternative to verifying the entire OS is to delegate critical functionality to special low-level execution platforms such as hypervisors, separation kernels, or micro-kernels. Such an approach has some significant advantages. First, the size and com-

plexity of the execution platform can be made much smaller, potentially opening up for rigorous verification. The literature has many recent examples of this, in seL4 [33], Microsoft’s Hyper-V project [34], Green Hills’ CC certified INTEGRITY-178B separation kernel [42], and the Singularity [29] microkernel. Second, the platform can be opened up to public scrutiny and certification, independent of application stacks.

Virtualization-like mechanisms can also be used to support various forms of application hardening against untrusted OSs. Examples of this include KCoFi [14] based on the Secure Virtual Architecture (SVA) [17], Overshadow [10], Inktag [28], and Virtual Ghost [15]. All these examples rely crucially on memory isolation to provide the required security guarantees, typically by virtualizing the memory management unit (MMU) hardware. MMU virtualization, however, can be exceedingly tricky to get right, motivating the use of formal methods for its verification.

In this paper we present an MMU virtualization API for the ARMv7-A processor family and its formal verification down to the binary level. A distinguishing feature of our design is the use of direct paging, a virtualization mechanism introduced by Xen [7] and used later with some variations by the SVA. In direct paging, page tables are kept in guest memory and allowed to be read and directly manipulated by the untrusted guest OS (when they are not in active use by the MMU). Xen demonstrated that this approach has better performance than other software virtualization approaches (e.g. shadow page tables) on the x86 architecture. Moreover, since direct paging does not require shadow data structures, this approach has small memory overhead. The engineering challenge inherent to this project is to design a minimal API that (i) is sufficiently expressive to host a paravirtualized Linux, (ii) introduces an acceptable overhead and (iii) whose implementation is sufficiently small to be subject to pervasive verification for a commodity CPU architecture such as ARMv7.

The security objective is to allow an untrusted guest system to operate freely, invoking the hypervisor at will, without being able to access memory or processor resources that the guest has not received static permission for. In this paper we describe the design, implementation, and evaluation of our memory virtualization API, and the formal verification of its security properties. The verification is performed using a formal model of the ARMv7 architecture [22], implemented in the HOL4 interactive theorem prover.

The proof strategy is to establish a bisimilarity between the hypervisor executing on a formal model of the ARMv7 instruction set architecture and the top level specification (TLS). The TLS describes the desired behaviour of the system consisting of handlers implementing the virtualization mechanism and the behaviour of machine instructions executed by the untrusted guest. The specification of the MMU virtualization API involves an abstract model state that is not represented in memory and thus by design invulnerable to direct guest access. Due to the direct paging approach, however, the page tables that control the MMU are residing in guest memory and need to be modelled explicitly. Hence, it is no longer self-evident that the desired memory isolation properties, no-exfiltration and no-infiltration in the terminology of [27], hold for guests in the TLS, and an important and novel part of the verification is therefore to formally validate that these properties indeed hold.

To keep the TLS as simple and abstract as possible, the TLS addresses page tables directly using their physical addresses. A real implementation cannot do this, but must

use virtual addresses instead, in addition to managing its internal data structures. To this end an implementation model is introduced, which uses virtual addresses instead of physical ones and stores the abstract model state explicitly in memory. This provides a very low-level C-like model of handler execution, directly reflecting all algorithmic features of the memory subsystem virtualization implemented by the binary code of the handlers, on the real ARMv7 state, as represented by the HOL4 model. We exhibit a refinement from the TLS to the implementation model, prove its correctness, and show, as a corollary, that the memory isolation properties proved for the TLS transfer to the implementation model. This constitutes the second part of the verification.

The next step is to fill the gap between the verification of this low-level abstraction and the binary level. To accomplish this an additional refinement must be established. Using the same approach as [20], we demonstrate how this can be achieved using a combination of theorem proving and tools that check contracts for binary code. The machine code verification is then in charge of establishing that the hypervisor code fragments respect these contracts, expressed as Hoare triples. Pre and post conditions are generated semi-automatically starting from the specification of the low-level abstraction and the refinement relation. They are then transferred to the binary analysis tool BAP [9], which is used to verify the hypervisor handlers at the assembly level. Several tools have been developed to support this task, including a lifter that transforms ARM code to the machine independent language that can be analysed by BAP and a procedure to resolve indirect jumps. The binary verification of the hypervisor has not been completed yet. However, we demonstrate the methodology outlined above by applying it to prove correctness of the binary code of one of the API calls. The scalability of the approach has been shown in [19], where it was used to verify the binary code of a complete separation kernel.

An alternative approach would be to focus the code verification at the C level. First, such an approach does not directly give assurances at the ISA level, which is our objective. This can be partly addressed by a certifying compiler such as CompCert[35]. However, system level code is currently not supported by such compilers. Moreover, this type of code is prone to break the standard C-semantics, for example by reconfiguring the MMU and changing the virtual memory mapping of the program under verification as is the case here.

The verification highlighted three classes of bugs in the initial design of the virtualization mechanism:

1. Arithmetic overflows, bit field and offset mismatches, and signed operators where the unsigned ones were needed.
2. Missing checks of self referencing page tables.
3. Approval of guest requests that cause unpredictable behaviours of the ARMv7 MMU.

Moreover, the verification of the implementation model identified additional bugs exploitable by requesting the validation of physical blocks residing outside the guest memory. This last class of bugs was identified because the implementation model takes into account the virtual memory mapping used by the handlers. Finally, the binary code verification identified a buffer overflow.

We report on a port of Linux kernel 2.6.34 and demonstrate the prototype implementation of a hypervisor for which the core component is the verified MMU virtualization API. The complete hypervisor augments the memory virtualization API by handlers that route aborts and interrupts inside Linux. Experiments demonstrate that the hypervisor can run with reasonable performance on real hardware (Beagleboard-xM based on the Cortex-A8 CPU). Furthermore an application scenario is demonstrated based on a trusted run-time monitor. The monitor executes alongside the untrusted Linux system, enforces the $W \oplus X$ policy (no memory area can be writable and executable simultaneously) and uses code signing to prevent binary code injection in the untrusted system.

1.1 Scope and limitations

The binary verification of the hypervisor has not been completed yet. However, we demonstrate the methodology outlined above by applying it to prove correctness of the binary code of one of the API calls. The scalability of the approach has been shown in [19], where it was used to verify the binary code of a complete separation kernel. In Section 9.5 we comment on the tasks that are not automated and need to be manually accomplished to complete the verification.

2 Related Work

The size and complexity of commodity OSs make them susceptible to attacks that can bypass their security mechanisms, as demonstrated in e.g. [47, 31]. The ability to isolate security critical components from an untrusted OS allows non critical parts of a system to be implemented while the critical software remains adequately protected. This isolation can be used both to protect applications from an untrusted OS as well as to protect the OS itself from internal threats. For example, KCoFI [14] uses Secure Virtual Architecture [17] to isolate the OS from a run-time checker. The checker instruments the OS and monitors its activities to guarantee the control-flow integrity of the OS itself. Related examples are application hardening frameworks such as Overshadow [10], Inktag [28], and Virtual Ghost [15]. In all these cases some form of virtualization of the MMU hardware is a critical component to provide the required isolation guarantees.

Shadow page tables (SPT) is a common approach to MMU virtualization. The virtualization layer maintains a shadow copy of page tables created and maintained by the guest OS. The MMU uses only the shadow pages, which are updated after the virtualization layer validates the OS changes. The Hyper-V hypervisor which uses shadow pages on x86, has been formally verified using the semi automated VCC tool [34]. Related work [5, 41] uses shadow page tables to provide full virtualization, including virtual memory, for “baby VAMP”, a simplified MIPS, using VCC. This work, along with later work [4] on TLB virtualization for an abstract mode of x64, has been verified using Wolfgang Paul’s VCC-based simulation framework [13]. Also, the OKL4-microvisor uses shadow paging to virtualize the memory subsystem [26]. However, this hypervisor has not been verified.

Some modern CPUs provide native hardware support for virtualization. The ARM Virtualization Extensions [3] augment the CPU with a new execution mode and provide a two stage address translation. These features greatly reduce the complexity of the virtualization layer [48]. XHMF [49] and CertiKOS [24] are examples of verified hypervisors for the x86 architecture that control memory operations of guests using virtualization extensions. The availability of hardware virtualization extensions, however, does not make software based solutions obsolete. For example, the recent Cortex-A5 (used in feature-phones) and the legacy ARM11 cores (used in home network appliances and the 2014 “New Nintendo 3DS”) do not make use of such extensions. Today, the Internet of Things (IoT) and wearable computing are dominated by microcontrollers (e.g. Cortex-M). As the recent Intel Quark demonstrates, the necessity of executing legacy stacks (e.g. Linux) is pushing towards equipping these microcontrollers with an MMU. Quark and the upcoming ARMv8-R both support an MMU and lack two stage page-tables. Generally, there is no universal sweet spot that reconciles the demands for low cost, low power consumption and rich hardware features. For instance, solutions based on FPGAs and soft-cores such as LEON can benefit from software based virtualization by freeing gates not used for virtualization extensions to be used for application specific logic (e.g. digital signal processing, software-defined radio, cryptography).

A virtualization layer provides to the guest OS an interface similar to the underlying hardware. An alternative approach is to execute the commodity OS as a partition of a microkernel, by mapping the OS threads directly to the microkernel threads, thus delegating completely the process management functionality from the hosted OSes to the microkernel (e.g. L⁴Linux). This generally involves an invasive and error-prone OS adaptation process, however. The formal verification of seL4 [33] demonstrated that a detailed analysis of the security properties of a complete microkernel is possible even at the machine code level [44]. Similarly, the Ironclad Apps framework [25] hosts security services in a remote operating system. Its functional correctness and information flow properties are verified on the assembly level.

In order to achieve trustworthy isolation between partitions, more light-weight solutions can also be employed, namely formally verified separation kernels [42, 19, 8] and Software Fault Isolation (SFI) [50, 52]. The latter has the advantage over the former in that it is a software-only approach, not relying on common hardware components such as MMU and memory protection units (MPU). Nevertheless, both mechanisms are generally not equipped with the functionality needed to host a commodity OS. Conversely, formally verified processor architectures specifically designed with a focus on logical partitioning [51] and information flow control [6] can be used to achieve isolation.

2.1 Contributions.

We present a platform to virtualize the memory subsystem of a real commodity CPU architecture: The ARMv7-A. The virtualization platform is based on direct paging, a virtualization approach inspired by the paravirtualization mechanism of Xen [7] and Secure Virtual Architecture [17]. The design of the platform is sufficiently slim to enable its formal verification without penalizing the system performance. The verification is performed down to a detailed model of the architecture, including a detailed model of the ARMv7 MMU. This enables our threat model to consist of an arbitrary

guest that can execute any ARMv7 instruction in user mode. We prove complete mediation of the MMU configurations, memory isolation of the hosted components, and information flow correctness. Additionally, we present our methodology for the binary verification of hypervisor code and report on first results. So far, one handler has been verified on the binary level. Completing the binary verification for all handlers is work in progress. The viability of the platform is demonstrated via a prototype hypervisor that is capable of hosting a Linux system while provably isolating it from other services. The hypervisor supports BeagleBoard-xM (a development board based on ARM Cortex-A8) and is used to benchmark the platform on real hardware. As the main application it is shown how the virtualization mechanism can be used to support a tamper-proof run-time monitor that prevents code injection in an untrusted Linux guest.

3 Verification Approach

In Figure 1 we give an overview of the entire verification flow presented in this paper. In particular it depicts the different layers of modelling, how they are related, and the tools used. This is discussed in more detail in Section 3.6.

Our MMU virtualization API is designed for paravirtualization and targets a commodity CPU (ARMv7-A). In such a scenario, the hosting CPU must provide two levels of execution: privileged and unprivileged. The hypervisor is the only software component that is executed at the privileged level; at this level the software has complete control of the underlying hardware. All other software components (including operating system kernels, user processes, etc.) are executed in unprivileged mode; direct accesses to the sensitive resources must be prevented and all transitions to privileged mode are controlled through the use of exceptions and interrupts.

In addition to the MMU virtualization API itself, as part of the hypervisor, the system is intended to support two types of clients:

- An untrusted commodity OS guest (Linux) running non-critical software (e.g. GUI, browser, server, games).
- A set of trusted services such as controllers that drive physical actuators, run-time monitors, sensor drivers, or cryptographic services.

An example computation of such system is shown in the row labelled “Real model” of Figure 1. White circles represent states in unprivileged execution level where the untrusted guest (either its kernel or one of its user processes) are running. Gray circles represent unprivileged states where one of the trusted services are in control. Finally, black circles represent states in privileged level where the hypervisor is active. Transitions between two unprivileged states (e.g. $1 \rightarrow 2$) do not cause any exceptions. The transition between the states 2 and 3 is caused by an exception, for example the execution of a software interrupt. Finally, transitions from privileged to unprivileged levels (e.g. $6 \rightarrow 7$) are caused by instructions that explicitly change the execution level.

Figure 1: Executions of a real machine (middle), the implementation model (above), and the Top Level Specification (top) and the relations between them. In addition the dependencies of the binary verification methodology (bottom) are depicted.

3.1 Attack Model

Due to the size and complexity of a complete Linux system, a realistic adversary model must consider the Linux partition compromised. For this reason, the attacker is an untrusted paravirtualized Linux kernel and its user processes, that maliciously or due to an error may attempt to gain access to resources outside the guest partition. Thus, the attacker is free to execute any CPU instruction in unprivileged mode; it is initially not able to directly access the coprocessor registers, and all attacker memory accesses are initially mediated by the MMU. However, by exploiting possible flaws in the hypervisor the attacker may during the course of a computation gain such access to the MMU configuration, something our security proof shows is in fact not possible. In this work, we assume absence of external mechanisms that can directly modify the internal state of the machine (e.g. external devices or physical tampering). The analysis of temporal partitioning properties (e.g. timing channels as investigated in [12]) is also deliberately left out of this work.

3.2 Security Goals

The verification must demonstrate that the low level platform does not allow undesired interference between guest and sensitive resources. That is:

1. The hypervisor must play the role of a security monitor of the MMU settings. If *complete mediation* of the MMU settings is violated, then an attacker may bypass the hypervisor policies and compromise the security of the entire system. We show this by proving that neither the untrusted guest nor the trusted services can directly change the MMU configuration.
2. Executions of an arbitrary guest cannot affect the “trusted world”, i.e. the parts of the state the guest is not allowed to modify, such as memory of trusted ser-

vices, system level registers and status flags, and the hypervisor state. This is an integrity property, similar to the no-exfiltration property of [27].

3. Absence of information flow from the trusted world to the guest, i.e. confidentiality, similar to no-infiltration of [27].

Note that these properties, as in [27], are qualitatively different: The integrity property is a single-trace property, and concerns the inability of the guest to directly write some other state variables. Since it is under guest control when and how to invoke the virtualization API, there are plenty of indirect communication channels connecting guests to the hypervisor. For instance, a guest decision to allocate or deallocate a page table affects large parts of the hypervisor state, without ever directly writing to any internal hypervisor state variable. Enforcing this is in a sense the very purpose of the hypervisor. On the other hand, the only desired effects of hypervisor actions should be to allocate/deallocate, map, remap, and unmap virtual memory resources, leaving any other observation a guest may make unaffected, thus preventing the guest from extracting information from inaccessible resources even indirectly. This is essentially a two-trace information flow property, needed to break guest-to-guest (or guest-to-service) information channels in much the same way as intransitive noninterference is used in [38] to break guest-to-guest channels passing through the scheduler in seL4.

In this work we establish these properties via successive refinements that add more details (that in turn can highlight different misbehaviour of the system) to the virtualization API, starting from an abstract model refining down to the binary code of the low level execution platform. We first demonstrate that the intended security property holds for the most abstract model. At each refinement, the proof consists of (i) identifying a relation that is strong enough to transfer the security property from the higher abstract model to the more real one (we call this a *candidate relation*) and (ii) demonstrating that the candidate relation actually satisfies the properties required from a refinement relation. For the first task it turns out that one needs a bisimulation relation in order to transfer higher-order information flow properties like confidentiality. The latter task is reduced to subsidiary properties, which have natural correspondences in previous kernel verification literature [27, 42]:

- A malicious guest cannot violate isolation while it is executing.
- Executions of the abstract vs the more real model preserve the candidate bisimulation relation.

These two tasks are qualitatively different. The former task, due to our use of memory protection, is really a noninterference-like property of the hosting architecture rather than a property of the hypervisor. This property must hold independently of the hosted guest, which is unknown at verification time since the attacker can take complete control of the untrusted Linux. By contrast, the latter task consists in verifying at increasing levels of detail the functional correctness of the individual handlers.

3.3 Top Level Specification

The first verification task focuses on establishing correctness of the design of the virtualization API. With this goal, in Section 6.1 we specify the desired behaviour of the

virtualization API as a transition system, called the *Top Level Specification* (TLS). This specification models unprivileged execution of an arbitrary guest system on top of a CPU with MMU support, alternating with abstract handler events. These events model invocations of the hypervisor handlers as atomic transformations operating on an abstract machine state. Abstract states are real CPU states extended by auxiliary (model) data that reflect the internal state of the hypervisor. We refer to this auxiliary data as the abstract hypervisor state. Handler events represent the execution of several instructions at privileged level, in response to exceptions or interrupts. Modelling handler effects as atomic state transformations is possible, since the hypervisor is non-preemptive, i.e. nested exceptions/interrupts are ruled out by the implementation.

Since in direct paging the guest systems can directly manipulate inactive page tables, the TLS needs to explicitly model page tables in memory. This contrasts simpler models such as the one presented in [19] where the hypervisor state was represented in the TLS using abstract model variables only. For this reason, establishing complete mediation, integrity, and confidentiality for the TLS is far from trivial.

3.4 Implementation Model

Extending the security properties to an actual implementation, however, requires additional work, for the following reasons:

- The TLS uses auxiliary data structures (the abstract hypervisor state) that are not stored inside the system memory.
- The TLS accesses the memory directly using physical addresses.

As is common practice, the virtualization code executes under the same address translation as the guest (but with different access permissions), in order to reduce the number of context switches required. For this approach it is critical to verify that all low-level operations performed by the hypervisor correctly implement the TLS specification; these operations include reads and updates of the page tables, and reads and updates of the hypervisor data structures. To show implementation soundness we exhibit a refinement property relating TLS states with states of the implementation. The refinement relation is proven to be preserved by all atomic hypervisor operations; reads and updates of the page tables, reads and updates of the hypervisor data structures. In particular it is established that these virtual memory operations access the correct physical addresses and never produce any data abort exceptions. Moreover, it is shown that the refinement relation directly transfers both the integrity properties and the information flow properties of the TLS to the implementation level.

3.5 Binary Verification

The last verification step consists in filling the gap between the implementation and the binary code executed on the actual hardware. This requires to exhibit a refinement relation between the implementation model and the real model of the system (i.e. where each transition represents the execution of one binary instruction).

Intuitively, internal hypervisor steps cannot be observed by the guests, since during the execution of the handler no guest is active. Moreover, as the hypervisor does not support preemption, then the execution of handlers cannot be interrupted. These facts permit to disregard internal states of the handlers and limit the refinement to relate only states where the guests are executing.

Thus, the binary verification can be accomplished in three steps: (i) verification that the refinement relation directly transfers both the isolation properties to the real model, (ii) verification of a top level theorem that transforms the relational reasoning into a set of contracts for the handlers and guarantees that the refinement is established if all contracts are satisfied, and (iii) verification of the machine code. The last step establishes if the hypervisor code fragments respect the contracts, expressed as Hoare triples $\{P\}C\{Q\}$, where P and Q are the pre/post conditions of the assembly fragment C .

3.6 Proof Engineering

We use Figure 1 and Table 1 to summarise the models, theorems and tools that are described in the following sections. We use three transition systems; the TLS (Section 6.1), the Implementation Model (Section 6.2) and the ARMv7 model (Section 4). These transition systems have been defined in the HOL4 theorem prover and differ in the level of abstraction they use to represent the hypervisor behaviour. The three transition systems model guest behaviour identically (e.g. transitions $0 \rightarrow 1$); these transitions obey the access privileges computed by the MMU and satisfy properties 1 and 2 of Section 4. These properties have been verified for a simplified MMU model in [32].

We use HOL4 to verify that the security properties hold for the TLS (Theorems 1, 2, 3, 4 and 5 of Section 6.1). The reasoning used to implement the proofs in the interactive theorem prover is summarised in Section 7.

The refinement ($\mathcal{R}$) between the TLS and the implementation model is verified in HOL4 (Theorem 6 of Section 6.2). We also use HOL4 to prove that the refinement transfers the security properties of the TLS to the implementation model (Corollary 1).

The refinement ($\mathcal{R}'$) between the implementation model and the real model is formally defined in HOL4, allowing us to prove that the refinement transfers the security properties to the ARMv7 model (Corollary 2).

The verification of the refinement (Theorem 7 of Section 6.3) is only partial: we demonstrate the verification of the binary code of the hypervisor only for a part of the code-base and we rely on some assumptions in order to fill the semantic gap between HOL4 and the external tools. We prove Theorem 7 for non-privileged transitions in HOL4 (i.e. transitions not involving the hypervisor code such as $1 \rightarrow 2$ and $12 \rightarrow 13$).

For the hypervisor code, we show that the task can be partially automated by means of external tools. For this purpose we use the HOL4 model of ARMv7 to transform the binary code of the hypervisor (e.g. the code executed between states 3 and 7 in the real model) to the input language of BAP (represented in the figure by arrow labelled “Lifter”). The usage of HOL4 for this task allows us to reduce the assumptions needed to fill the gap between the HOL4 ARMv7 model and BAP, as described in

Section 9. The methodology to complete the verification is the following: given a hypervisor handler whose code has been translated to the BAP code C , we use a HOL4 certifying procedure that generates a contract $\{P\}C\{Q\}$ starting from the hypervisor implementation model and the refinement relation. The certifying procedure yields a HOL4 theorem stating that the refinement relation $\mathcal{R}'$ is preserved if the hypervisor handler C establishes the postcondition Q starting from the precondition P . We use BAP to compute the weakest precondition WP of the postcondition Q and the code C and a finally an SMT solver checks that the weakest precondition is entailed by the precondition.

4 The ARMv7 CPU

ARMv7 is the currently dominant processor architecture in embedded devices. Our verification relies on the HOL4 model of ARM developed at Cambridge [22]. The use of a theorem prover allows the verification goals to be stated in a manner which is faithful to the intuition, without resorting to approximations and abstractions that would be needed when using a fully automated tool such as a model checker. Furthermore, basing the verification on the Cambridge ARM model lends high trustworthiness to the exercise: The Cambridge model is well-tested and phrased in a manner that retains a high resemblance to the pseudocode used by ARM in the architecture reference manual [1]. The Cambridge model has been extended by ourselves to include MMU functionality. The resulting model gives a highly detailed account of the ISA level instruction semantics at the different privilege levels, including relevant MMU coprocessor effects. It must be noted that the Cambridge ARM model assumes linearizable memory, and so can be used out of the box only for processor and hypervisor implementations that satisfy this property, for instance through adequate cache flushing as discussed in Section 5.5.

We outline the HOL4 ARMv7 model in sufficient detail to make the formal results presented later understandable. An ARMv7 machine state is a record

$$\sigma = \langle uregs, bregs, coregs, mem \rangle \in \Sigma,$$

where *uregs*, *bregs*, *coregs*, and *mem*, respectively, represent the user registers, banked registers (used for handling exceptions), coprocessors, and memory. The function $mode(\sigma)$ returns the current privilege execution mode in the state σ , which can be either *PL0* (unprivileged or user mode, used by the guest) or *PL1* (privileged mode, used by the hypervisor). The memory is the function $mem \in 2^{32} \rightarrow 2^8$. The coprocessor registers *coregs* control the MMU.

System behaviour is modelled by the state transition relation $\rightarrow_{l \in \{PL0, PL1\}} \subseteq \Sigma \times \Sigma$, where a transition is performed by the execution of an ARM instruction. Unprivileged transitions ($\sigma \rightarrow_{PL0} \sigma'$) start from and end in states that are in unprivileged execution mode (i.e. $mode(\sigma) = mode(\sigma') = PL0$). All the other transitions ($\sigma \rightarrow_{PL1} \sigma'$) involve at least one state in privileged level. The raising of an exception is modelled by a transition that enables the level *PL1*. An exception can be raised because: (i) a software interrupt (SWI) is executed, (ii) the current instruction is undefined, (iii) a memory access is attempted that is disallowed by the MMU, or (iv) an

Artefact	Description	HOL4	BAP	Appearance
TLS	Model of the abstract design of the hypervisor + attacker (guest)	◦		[40]
Implementation model	Low level model of the hypervisor + attacker	◦		[20]
ARM model	Real model of the system	◦		[22, 40]
Properties 1 and 2	Properties of the ARM instruction set (here only assumed)	◦		[32]
Properties of the TLS				
Theorem 1	Verification of the functional invariant	◦		[40]
Theorem 2	Verification of MMU integrity	◦		[40]
Theorem 3	Verification of no context switch	◦		[40]
Theorems 4 and 5	Verification of No exfiltration + No infiltration = isolation	◦		[40]
Properties of the Implementation model				
Theorem 6	Verification of Refinement	◦		[20]
Corollary 1	Verification of MMU integrity + No exfiltration + No infiltration	◦		[20]
Properties of the real model				
Theorem 7	Refinement. For non-privileged transitions proved in HOL4. One of the API function proved using BAP	◦	◦	Here
Corollary 2	Verification of MMU integrity + No exfiltration + No infiltration	◦		Here
Miscellaneous				
Lifter	Translation of ARMv7 binary to BIL	◦	◦	[20, 19]
Certifying procedure	Generates a contract starting from the model of one of the API function and the refinement relation	◦		Here
Indirect jump solver	Computes all possible target of indirect jumps for a BIL loop free program. Here extended and re-implemented as BAP extension		◦	[20]

Table 1: List and first appearance of models, theorems and tools.

hardware interrupt is received. Whenever an exception occurs, the CPU disables the interrupts and jumps to a predefined address in the vector table to transfer control to the corresponding exception handler.

The ARMv7 MMU uses a two level translation scheme. The first level (L1) consists of a 4096 entry table that divides up to 4GB of memory into 1MB sections. These sections can either point to an equally large region of physical memory or to a level 2 (L2) page table with 256 entries that maps the 1MB section into 4 KB physical pages. MMU behaviour is modelled by the function $mmu(\sigma, pl, va, req)$, which takes a state σ , a privilege level, a virtual address va and an access request $req \in \{rd, wt, ex\}$ (representing read, write and execute accesses) and yields $pa \in 2^{32} \cup \{\perp\}$, where pa is the translated physical address or an access denied. The ARMv7 documentation describes the possibility of unpredictable behaviour due to erroneous setup of the MMU through coprocessor registers and page tables. In this work the hypervisor completely mediates the MMU configuration and aims to rule out this kind of behaviour.

In the ARM architecture *domains* provide a discretionary access control mechanism. This mechanism is orthogonal to the one provided by CPU execution modes. There are sixteen domains, each on activated independently in one of the coprocessor registers *coregs*. The page tables map each virtual page/section to one of the domains and the MMU forbids accesses to a page/section if the corresponding domain is not active.

The state transition relation queries the MMU whenever a virtual address is accessed, and raises an exception if the requested access mode is not allowed. To describe the security properties guaranteed by an ARMv7 CPU we introduce some auxiliary definitions.

Definition 1 (Physical memory access rights) *The predicate mmu_{ph} takes a state σ , the privilege level pl , a physical address pa and an access permission $req \in \{rd, wt, ex\}$ and holds if the access permission is granted for physical address pa .*

$$mmu_{ph}(\sigma, pl, pa, req) \Leftrightarrow \exists va. mmu(\sigma, pl, va, req) = pa$$

The ARMv7 MMU mediates accesses to the virtual memory, enabling or forbidding operations to virtual addresses. Intuitively, a physical address pa can be read (written) if it exists at least a virtual addresses va that can be read (written) and that is mapped to pa according with the current page tables.

Definition 2 (Write-derivability) *We say that a state σ' is write-derivable from a state σ in privilege level pl if their memories differ only for physical addresses that are writable in pl .*

$$wd(\sigma, \sigma', pl) \Leftrightarrow \forall pa. \sigma.mem(pa) \neq \sigma'.mem(pa) \Rightarrow mmu_{ph}(\sigma, pl, pa, wt).$$

According with the MMU configuration in σ , only a subset of physical addresses are writable ($mmu_{ph}(\sigma, pl, pa, wt)$). Write-derivability identifies the set of states that can be produced by changing the memory content of an arbitrary number of such physical addresses with arbitrary values.

Definition 3 (MMU-equivalence) We say that two states are *MMU-equivalent* if for any virtual address va the MMU yields the same translation and the same access permissions.

$$\sigma \equiv_{mmu} \sigma' \Leftrightarrow \forall va, pl, req. mmu(\sigma, pl, va, req) = mmu(\sigma', pl, va, req)$$

Informally, two states are *MMU-equivalent* if their MMUs are configured exactly in the same way.

Definition 4 (MMU-safety) Finally, a state is *MMU-safe* if it has the same MMU behaviour as any state with the same coprocessor registers whose memory differs only for addresses that are writable in *PL0*.

$$mmu_s(\sigma) \Leftrightarrow \forall \sigma'. \sigma.coregs = \sigma'.coregs \wedge wd(\sigma, \sigma', PL0) \Rightarrow (\sigma \equiv_{mmu} \sigma')$$

A state is *MMU-safe* if there is no way to change the MMU configuration (i.e. the page tables) by writing into addresses that writable in non-privileged mode. That is the MMU configuration prevent non-privileged SW to tamper the page tables.

An ARMv7 processor that obeys the access privileges computed by the MMU satisfies the following two properties:

Property 1 (ARM-integrity) Assume $\sigma \in \Sigma$ with $mode(\sigma) = PL0$. If $\sigma \rightarrow_{PL0} \sigma'$ and $mmu_s(\sigma)$ then $wd(\sigma, \sigma', PL0)$ and $\sigma.coregs = \sigma'.coregs$, i.e., *unprivileged steps from MMU-safe states can only lead into write-derivable states and do not affect the coprocessor registers.*

Note, that the MMU-safety prerequisite is not redundant here, because single instructions in ARM may result in a series of write operations, e.g., for “store pair” and unaligned store instructions. If the MMU configuration was not safe from manipulation in unprivileged mode, then such a series of writes could lead to an intermediate MMU configuration granting more write permissions than the initial one and the resulting state would not be write-derivable from σ .

Property 2 (ARM-confidentiality) Let $\sigma_1, \sigma_2 \in \Sigma$ with $mode(\sigma_1) = mode(\sigma_2) = PL0$, and let A contain all physical addresses accessible in σ_1 , i.e., $A \supseteq \{pa \mid \exists req. mmu_{ph}(\sigma_1, PL0, pa, req)\}$. Suppose that $\sigma_1.uregs = \sigma_2.uregs$, $\sigma_1.coregs = \sigma_2.coregs$, $\sigma_1 \equiv_{mmu} \sigma_2$, and $\forall pa \in A. \sigma_1.mem(pa) = \sigma_2.mem(pa)$. If $\sigma_1 \rightarrow_{PL0} \sigma'_1$, $\sigma_2 \rightarrow_{PL0} \sigma'_2$, $mmu_s(\sigma_1)$, and $mmu_s(\sigma_2)$ then

$$\sigma'_1.uregs = \sigma'_2.uregs, \sigma'_1.coregs = \sigma'_2.coregs \text{ and } \forall pa \in A. \sigma'_1.mem(pa) = \sigma'_2.mem(pa).$$

Intuitively, Property 2 establishes that in MMU-safe configurations unprivileged transitions only can access information stored in the registers and in the part of memory that is readable in *PL0* according to access permissions. Within this paper we take Properties 1 and 2 for granted. In [32] the authors validated the HOL4 ARMv7 model against these properties assuming an identity-mapped address translation. Extending the result for an arbitrary but MMU-safe page table setup is currently nearing completion.

5 The Memory Virtualization API

The memory virtualization API is designed for the ARMv7-A architecture¹ and assumes neither hardware virtualization extensions nor TrustZone [2] support. To properly isolate the trusted components from the untrusted guest, which hosts a commodity OS, the memory virtualization subsystem needs to provide two main functionalities:

- Isolation of memory resources used by the trusted components.
- Virtualization of the memory subsystem to enable the untrusted OS to dynamically manage its own memory hierarchy, and to enforce access restrictions.

The physical memory region allocated to each type of client is statically defined. Inside its own region the guest OS is free to manage its own memory, and the virtualization API does not provide any additional guarantees for the security of the guest OS kernel against attacks from its user processes. However, using trusted services such as a run-time monitor it is possible to provide provable security guarantees to the guest OS, for instance to enforce the $W \oplus X$ policy or to secure software updates, as explained in Section 12.

Figure 2: Direct paging: 1) guest writes to virtual memory are mediated by the MMU as usual; 2) page tables are allocated in guest memory; 3) the hypervisor prevents writable mappings to guest memory regions holding page tables, forbidding the guest to directly modify them; 4) the hypervisor allows writable mappings to data blocks in guest memory.

5.1 Memory Management

The virtual memory layout is defined by a set of page tables that reside in physical memory. The configuration of these page tables is security critical and must not be directly manipulated by untrusted parties. At the same time, the untrusted Linux kernel

¹In practice, the presented design also supports the ARMv6 and ARMv5 architectures.

<i>switch</i>	Select the active L1
<i>L1create</i>	Create page table of type L1
<i>L2create</i>	Create page table of type L2
<i>L1free</i>	Change the type of an L1 block to <i>data</i>
<i>L2free</i>	Change the type of an L2 block to <i>data</i>
<i>L1unmap</i>	Clear an entry of an L1 page table
<i>L2unmap</i>	Clear an entry of an L2 page table
<i>L1map</i>	Set an entry of an L1 page table
<i>L2map</i>	Set an entry of an L2 page table

Figure 3: The virtualization API of the hypervisor to support direct paging.

needs to manage its memory layout, which requires constant access to the page tables. Hence the hypervisor must provide a secure access mechanism, which we refer to as virtualizing the memory subsystem.

We use direct paging [7] to virtualize the memory subsystem. Direct paging allows the guest to allocate the page tables inside its own memory and to directly manipulate them while the tables are not in active use by the MMU. Once the page tables are activated, the hypervisor must guarantee that further updates are possible only via the virtualization API to modify, allocate and free the page tables.

Physical memory is fragmented into blocks of 4 KB. Thus, a 32-bit architecture has 2^{20} physical blocks. Since L1 and L2 page tables have size 16 KB and 1 KB respectively, an L1 page table is stored in four contiguous physical blocks and a physical block can contain four L2 page tables. We assign a type to each physical block, that can be:

- *data*: the block can be written by the guest.
- *L1*: contains part of an L1 and is not writable in unprivileged mode.
- *L2*: contains four L2 and is not writable in unprivileged mode.

The virtualization API shown in Figure 3 is very similar to the MMU interface of the Secure Virtual Architecture [17] and consists of 9 hypercalls that selects, creates, frees, maps, or unmaps memory blocks or page tables.

Figure 2 indicates the address translation procedure and the connection between components of memory subsystem.

5.2 Enforcing the page type constraints

Each API call needs to validate the page type, guaranteeing that page tables are write-protected. This is illustrated in Figure 4. The table in the centre represents the physical memory and stores the virtualization data structures for each physical block; the page type (*pt*), a flag informing if the block is allocated to guest memory (*gm*) and a reference counter (*rc*).

The four top most blocks contain an L1 page table, whose 4096 entries are depicted by the table *L1-A*. The top entry of the page table is a section descriptor ($T = S$) that

Figure 4: Direct-paging mechanism. We use solid arrows to represent the L2 page table references and unprivileged write permissions, dotted arrows to represent other allowed references, and dashed arrows for references violating the page table policy.

grants write permission to the guest ($AP = (0, w)$). This entry's address component (Adr) points to the second physical section, which consists of 256 physical blocks. Two more section descriptors of L1-A are depicted in the table: the first one grants read-only permission to the guest $(0, r)$, the second descriptor prevents any guest access and enables write permission for the privileged mode $(1, w)$. The last two entries of L1-A are PT-descriptors. Each entry points to an L2 page table in the same physical block described by table L2-A and containing four L2 page tables.

The API calls manipulating an L1 enforce the following policy:

Any section descriptor that allows the guest to access the memory must point to a section for which every physical block resides in the guest memory space. Moreover, if a descriptor enables a guest to write then each block must be typed *data*. Finally, all PT-descriptors must point to physical blocks of type L2.

The Figure depicts two additional L1 page tables; *L1-B* and *L1-C*. The type of a physical block containing *L1-B* can be transformed to *L1* by invoking *L1create*. On the other hand, a block containing *L1-C* is rejected by *L1create* since the block contains three entries that violate the policy. In fact,

- the first descriptor grants guest write permission over a section which has at least one non data block, in this case L2,
- the second section descriptor allows the guest to access a section of the physical memory in which there exists a block that is outside the guest memory, and
- the third entry is a PT-descriptor, but points to a physical block that is not typed L2.

The first setting clearly breaks MMU-safety, since the guest is now able to write directly to a page table, circumventing the complete mediation of MMU configurations by the

Figure 5: Spawning a new process using the virtualization API. The guest (1) requests a writable mapping of four physical blocks to allocate a new L1 page table. After (2) setting up the table in memory (not shown), it asks (3) to remove the writable mapping, (4) to interpret the blocks as an L1 table, and (5) to make this one the new active L1 page table (not shown).

hypervisor. The second situation compromises confidentiality and possible integrity of the system if the guest has write access to the block outside its own memory. The third issue may again break MMU-safety if the referenced block is a writable data block. In case the referenced block contains (part of) another L1 page table this setting can lead to unpredictable MMU behaviour, since the L1 page table entries have a different binary format than the expected L2 entries.

The table *L2-A* depicts the content of a physical block typed L2 that contains four L2 page tables, each consisting of 256 entries. Each hypercall that manipulates an L2 enforces the following policy:

If any entry of the four L2 page tables grants access permission to the guest then the pointed block must be in the guest memory. If the entry also enables guest write access then the pointed block must be typed *data*.

For example a block containing *L2-B* is rejected by *L2create*, since the block contains at least two entries that violate the policy and thus threaten MMU-safety and integrity (in case of the first entry shown) as well as confidentiality (in case of the second one).

A naive run-time check of the page-type policy is not efficient, since it requires to re-validate the L1 page table whenever the *switch* hypercall is invoked. To efficiently enforce that only *data* blocks can be written by the guest, the hypervisor maintains a reference counter, tracking for each block the sum of:

1. The number of descriptors providing writable access in user mode to the block.
2. The number of PT-descriptors that point to the block.

The intuition is that a hypercall can change the type of a physical block (e.g. allocate or free a page table) only if the corresponding reference counter is zero. Lemmas 5 and 6 in Section 7 demonstrate that this approach is sound and that the page table policy described above is sufficient to guarantee MMU-safety.

In Figure 5 we exemplify how an OS can use the API to spawn a new process. The OS selects four blocks from its physical memory to allocate a new L1 page table. We

assume that initially the OS has no virtual mapping that enables it to access this part of the memory (i.e. the reference counter *rc* of these blocks is zero and the type *pt* is *data*).

1. Using *L2map*, the OS requests to change an existing L2 page table, establishing a writable mapping to the four blocks. The hypercall increases the reference counter accordingly (i.e. $rc1 = 1$).
2. Without any mediation of the hypervisor, the OS uses the new mapping to write the content of the new L1 page table.
3. Using *L2unmap*, the guest removes the mapping established in (1) and decreases the reference counters (i.e. $rc3 = 0$).
4. The guest invokes *L1create*, requesting the page table to be validated and the block type changed to L1. The request is granted only if the reference counter is zero, guaranteeing that there does not exist any mapping in the system that allows the guest to directly write the content of the page table.
5. Finally, the OS invokes *switch* to perform the context switch and to activate the new L1.

The example demonstrates some of the principles used to design the minimal API: (i) the address of the page tables are chosen by the guest, thus we do not need to change the OS allocators, (ii) the preparation of the page table can be done by the OS without mediation of the hypervisor, (iii) the content of the page table is not copied into the hypervisor memory, thus reducing memory accesses and memory overhead and not requiring dynamic allocation in the hypervisor, (iv) tracking the reference counter is used to guarantee the absence of page tables granting the guest write access to another page table, thus we can allow context switches among all created L1s without needing to re-validate their content.

5.3 Integrity of the Hypervisor Memory Map

When an exception is raised, the CPU redirects execution flow to a fixed location according to the exception vector. In ARMv7, subsequent instructions are executed in privileged mode but under the same virtual memory mapping as the interrupted guest. The hypervisor must enforce that the memory mapping of the exception vector, handler code, and hypervisor data structures is accessible during an exception without being modifiable by the guests. To this end, the hypervisor maintains its own static virtual memory mapping in a master page table and mirrors the corresponding regions to all L1s of the guest (with restricted access permissions).

5.4 Hypervisor Accesses to Guest Page Tables.

The hypervisor APIs must be able to read and write the page tables allocated by the guest, in order to check the soundness of the requests and to apply the corresponding changes. The naive solution requires the hypervisor to change the current page table,

enabling a hypervisor master page table whenever the guest memory must be accessed and then re-enabling the original page table before the guest is restored. This solution is expensive as it requires to flush TLB and caches. A solution tailored for Unixes can rely on the injective mapping built by the guest, which can be used by the hypervisor to access the guest kernel memory. In our settings the hosted guest is not trusted, thus this solution cannot guarantee that the injective mapping is obeyed by the guest. Some ARMv7 CPUs support special coprocessor instructions for virtual-to-physical address translation. These instructions can be used to validate the guest injective mapping at run-time. However, this approach is platform dependent and can result in nested exceptions that complicate the architecture and verification of the hypervisor. Instead, our design reserves a subset of the virtual address space for hypervisor use. The hypervisor master page table is built so that this address space is always mapped according to an injective translation (1-to-1) allowing the hypervisor to easily compute the virtual address for each physical address in the guest memory, similar to the direct memory maps supported by FreeBSD [36] and Linux [21]. As with the hypervisor code and data structures, these regions are mirrored in all guest L1 tables.

5.5 Memory Model and Cache Effects.

Hypervisors are complex software interacting directly with many low level hardware components, like processor, MMU, etc. Furthermore, there are hardware pieces that, while being invisible to the software layer, still can affect the system behaviour in many aspects. For example, the memory management unit relies on a caching mechanism, which is used to speed up accesses to page table entries. Basically, a data-cache is a shared resource between all partitions and it thus affects and is affected by activities of each partition. Consequently, data-caches may cause unintended interaction between software components running on the same processor, which can lead to cross-partition information leakage.

Moreover, for the ARMv7 architecture cache usage may cause sequential consistency to fail if the same physical address is accessed using different cacheability attributes. This opens up for TOCTTOU²-like vulnerabilities since a trusted agent may check and later evict a cached data item, which is subsequently substituted by an unchecked item placed in the main memory using an uncacheable alias. Furthermore, an untrusted agent can similarly use uncacheable address aliasing to easily measure which lines of the cache are evicted. This results in storage channels that are not visible in information flow analyses performed at the ISA level.

As an example (Figure 6), the guest can use an uncacheable virtual alias of a page table entry in physical memory to bypass the page type constraints and install a potentially harmful page table. If the cache contains a valid page table entry PTE A for the physical address from a previous check by the hypervisor and this cache entry is clean (i.e., it will not be written back to memory upon eviction), the guest can (1) store an invalid (i.e. violating the page table policy) page table entry PTE B in a data page and (2) request the data page to become a page table. If the guest write is (3) directly applied to the memory, bypassing the cache using an uncacheable virtual address, and (4) the

²TOCTTOU – Time Of Check To Time Of Use

Figure 6: Integrity threat due to incoherent memory caused by mismatched cacheability attributes. PTE A is validated by the hypervisor (4) but PTE B will be used as a page table entry for the guest (6).

hypervisor accesses the same physical location through the cache, then the hypervisor potentially validates stale data (5). At a later point in time, the validated data PTE A is evicted from the cache and not written back to memory since it is clean. Then (6) the MMU will use the invalid page table containing PTE B instead and its settings become untrusted.

This kind of behaviour undermines the properties assured by formal verification that assumes a sequentially consistent model. In this paper, to counter this threat we use a naive solution; we prevent memory incoherence by cleaning the complete cache before accessing data stored by the guest. Clearly, this can introduce a substantial performance overhead, as shown in Section 10.2. In [18], we demonstrate more efficient countermeasures to such threats and propose techniques to fix the verification.

6 Formalizing the Proof Goals

6.1 The Top Level Specification

A state of the Top Level Specification (TLS) is a tuple $\langle \sigma, h \rangle$, consisting of an ARMv7 state σ and an abstract hypervisor state h . An abstract hypervisor state has the form $\langle pgtype, pgrefs \rangle$ where $pgtype$ indicates memory block types and $pgrefs$ maintains reference counters. Specifically, $pgtype \in 2^{20} \rightarrow \{D, L1, L2\}$ tracks the type of each 4 KB physical block; a block can either be (D) memory writable from the guest or data page, (L1) contain a L1 page table or (L2) contain a L2 page table. The map $pgrefs \in 2^{20} \rightarrow 2^{30}$ tracks the references to each physical block, as described in Section 5.

The TLS interleaves standard unprivileged transitions with abstract handler invocations. Formally, the TLS transition relation $\langle \sigma, h \rangle \rightarrow_{i \in \{0,1\}} \langle \sigma', h' \rangle$ is defined as follows:

- If $\sigma \rightarrow_{PLO} \sigma'$ then $\langle \sigma, h \rangle \rightarrow_0 \langle \sigma', h \rangle$; instructions executed in unprivileged mode that do not raise exceptions behave equivalently to the standard ARMv7 semantics and do not affect the abstract hypervisor state.

- If $\sigma \rightarrow_{PL1} \sigma'$ and $mode(\sigma) = PL0$ then $\langle \sigma, h \rangle \rightarrow_1 H_a(\langle \sigma', h \rangle)$; whenever an exception is raised, the hypervisor is executed, modelled by the abstract handler H_a . The abstract handler always yields a state whose execution mode is unprivileged.

In our setup the trusted services and the untrusted guest are both executed in unprivileged mode. To distinguish between these two partitions, we use ARM domains. We reserve the domains 2-15 for the secure services.

Definition 5 (Secure services) *Let $\sigma \in \Sigma$, the predicate $S(\sigma)$ identifies if the active partition is the one hosting the secure services: the predicate holds if at least one of the reserved domains (2-15) is enabled in the coprocessor registers $coregs$ of σ .*

Intuitively, guaranteeing spatial isolation means confining the guest to manage a part of the physical memory available for the guest uses. In our setting, this part is determined statically and identified by the set of physical addresses G_m . Clearly, no security property can be guaranteed if the system starts from an insecure state; for example the guest must not be allowed to change the MMU behaviour by directly writing the page tables. For this reason we introduce a system invariant $I(\langle \sigma, h \rangle)$ that is used to constrain the set of secure initial states of the TLS. The set of all possible TLS states that satisfy the invariant is denoted by $\mathcal{Q}_I$. Then one needs to show:

Theorem 1 (Invariant preserved) *Let $\langle \sigma, h \rangle \in \mathcal{Q}_I$ and $i \in \{0, 1\}$. If $\langle \sigma, h \rangle \rightarrow_i \langle \sigma', h' \rangle$ then $I(\langle \sigma', h' \rangle)$.*

Section 7 elaborates the definition of the invariant and summarises the proof of the Theorem.

Complete mediation (MMU-integrity) is demonstrated by showing that neither the guest nor the secure services are able to directly change the content of the page tables and affect the address translation mechanism.

Theorem 2 (MMU-integrity) *Let $\langle \sigma, h \rangle \in \mathcal{Q}_I$. If $\langle \sigma, h \rangle \rightarrow_0 \langle \sigma', h' \rangle$ then $\sigma \equiv_{mmu} \sigma'$.*

We use the approach of [27] to analyse the hypervisor data separation properties. The observations of the guest in a state $\langle \sigma, h \rangle$ is represented by the structure $O_g(\langle \sigma, h \rangle) = \langle \sigma.uregs, mem_g(\sigma), \sigma.coregs \rangle$ of user registers, guest memory $mem_g(\sigma)$ (the restriction of $\sigma.mem$ to G_m), and coprocessor registers. The latter are visible to the guest since they directly affect guest behaviour by controlling the address translation, and do not contain any information the guest should not be allowed to see. Evidently, however, all writes to the coprocessor registers must be mediated by the hypervisor.

The remaining part of the state (i.e. the content of the memory locations that are not part of the guest memory, banked registers) and, again, the coprocessor registers constitute the secure observations $O_s(\langle \sigma, h \rangle)$ of the state, which guest transitions are not supposed to affect.

The following theorem demonstrates that the context switch between the untrusted guest and the trusted services is not possible without the mediation of the hypervisor. The proof is straightforward, since S only depends on coprocessor registers that are not accessible in unprivileged mode.

Theorem 3 (No context switch) *Let $\langle \sigma, h \rangle \in \mathcal{Q}_I$. If $\langle \sigma, h \rangle \rightarrow_0 \langle \sigma', h' \rangle$ then $S(\sigma) = S(\sigma')$.*

The *no-exfiltration* property guarantees that a transition executed by the guest does not modify the secure resources:

Theorem 4 (No-Exfiltration) *Let $\langle \sigma, h \rangle \in \mathcal{Q}_I$. If $\langle \sigma, h \rangle \rightarrow_0 \langle \sigma', h' \rangle$ and $\neg S(\sigma)$ then $O_s(\langle \sigma, h \rangle) = O_s(\langle \sigma', h' \rangle)$.*

The *no-infiltration* property is a non-interference property guaranteeing that guest instructions and hypercalls executed on behalf of the guest do not depend on any information stored in resources not accessible by the guest.

Theorem 5 (No-Infiltration) *Let $\langle \sigma_1, h_1 \rangle, \langle \sigma_2, h_2 \rangle \in \mathcal{Q}_I$, $i \in \{0, 1\}$, and assume that $O_g(\langle \sigma_1, h_1 \rangle) = O_g(\langle \sigma_2, h_2 \rangle)$, $\neg S(\sigma_1)$, and $\neg S(\sigma_2)$.*

If $\langle \sigma_1, h_1 \rangle \rightarrow_i \langle \sigma'_1, h'_1 \rangle$ and $\langle \sigma_2, h_2 \rangle \rightarrow_i \langle \sigma'_2, h'_2 \rangle$ then $O_g(\langle \sigma'_1, h'_1 \rangle) = O_g(\langle \sigma'_2, h'_2 \rangle)$.

6.2 The Implementation Model

A critical problem of verifying low level platforms is that intermediate states of a handler that reconfigures the MMU can break the semantics of the high level language (e.g. C). For example, a handler can change a page table and (erroneously) unmap the region of virtual memory where the handler data structure (or the code) are located. For this reason we introduce the implementation model, that is sufficiently detailed to expose misbehaviour of the hypervisor accesses to the observable part of the memory (i.e. page tables, guest memory and internal data structures). The implementation interleaves standard unprivileged transitions and hypervisor functionalities. In contrast to the TLS, these functionalities now store their internal data in system memory, accessed by means of virtual addresses. In practice, in the implementation model the hypervisor functionalities are expressed as executable specifications, yet they are very close to the execution of the actual hardware at instruction semantics level. We demonstrate these differences by comparing two fragments of the TLS and the implementation specifications.

The TLS models the update of a guest page table entry as $\sigma'.mem = write_{32}(\sigma.mem, pa, desc)$, where pa is the physical address of the entry, $desc$ is a word representing the new descriptor and $write_{32}$ is a function that yields a new memory having four consecutive bytes updated. At the implementation level the same operation is represented as

```

if  $\neg mmu(\sigma, PL1, Gpa2va(pa), wt)$ 
  then  $\perp$ 
  else  $write_{32}(\sigma.mem,$ 
              $mmu(\sigma, PL1, Gpa2va(pa), wt), desc)$ 

```

where mmu is the formal model of the ARMv7 MMU (introduced in Section 4) and $Gpa2va$ is the function used by the hypervisor to compute the virtual address of a physical address that resides in guest memory. This function is statically defined and is the inverse of the injective translation established by the hypervisor master page table.

The implementation can fail to match the TLS for two reasons: (i) the current page table can prevent the hypervisor from accessing the computed virtual address, and then the implementation terminates in a failing state (denoted by $\perp$), (ii) the current address translation does not respect the expected injective mapping, thus $mmu(\sigma, PL1, Gpa2va(pa), wt) \neq pa$ and the implementation writes in an address that differs from the one updated by the TLS.

The next example shows the difference between accesses to the reference counter in the TLS and at implementation level. The TLS models this operation as $h.pgrefs(b)$, where b is the physical block. The implementation models the same operation using memory offsets as follows:

```

if  $\neg mmu(\sigma, PL1, tbl_{va} + 4*b, rd)$ 
then  $\perp$ 
else  $read_{32}(\sigma.mem,$ 
       $mmu(\sigma, PL1, tbl_{va} + 4*b, rd))$ 
       $\& 0xCFFFFFFF$ 

```

This representation is directly reflected in the hypervisor code. For each block, the page type (two bits) and the reference counter (30 bits) are placed contiguously in a word. These words form an array, whose initial virtual address is tbl_{va} .

The handlers are represented by a HOL4 function H_r from ARMv7 states to ARMv7 states. The function is the executable specification of the various exception handlers including the MMU mapping/remapping/unmapping functionalities and is composed sequentially of the functional specifications for the corresponding code segments.

Then, the state transition relation $\mapsto_{i \in \{0,1\}} \subseteq \Sigma \times (\Sigma \cup \{\perp\})$ determines the implementation behaviour as follows:

- If $\sigma \rightarrow_{PL0} \sigma'$ then $\sigma \mapsto_0 \sigma'$; instructions executed in unprivileged mode that do not raise exceptions behave according to the standard ARMv7 semantics.
- If $\sigma \rightarrow_{PL1} \sigma'$ and $mode(\sigma) = PL0$ then $\sigma \mapsto_1 H_r(\sigma')$; whenever an exception is raised, the hypervisor is executed and its behaviour is modelled by the function H_r . The function yields either a state whose execution mode is unprivileged or $\perp$.

To show implementation soundness we exhibit a refinement property relating abstract states $\langle \sigma_1, h \rangle$ to real states σ_2 . The refinement relation $\mathcal{R}$ (that is left-total and surjective with the exception of the faulty state $\perp$) requires that: (i) the registers and coprocessors contain the same value in both states, (ii) the guest memory contains the same values in both states, (iii) the hypervisor data structures of the TLS state are projected into a part of hypervisor memory, and (iv) the reserved virtual addresses are always mapped in the same way as they are mapped in the master page table. Observations of the guest O_g^r and secure observations O_s^r are defined on real states using the hypervisor data structure mapping in analogy with the corresponding observations on abstract states defined above. We require the refinement relation $\mathcal{R}$ to be a bisimulation relation, that is preserved by computations of the abstract and implementation model.

Theorem 6 (Implementation refinement) *Let $\langle \sigma_1, h \rangle \in \mathcal{Q}_I$ and $\sigma_2 \in \Sigma$ such that $\langle \sigma_1, h \rangle \mathcal{R} \sigma_2$. Let $i \in \{0, 1\}$.*

- If $\sigma_2 \mapsto_i \sigma'_2$ then exists $\langle \sigma'_1, h' \rangle$ such that $\langle \sigma_1, h \rangle \rightarrow_i \langle \sigma'_1, h' \rangle$ and $\langle \sigma'_1, h' \rangle \mathcal{R} \sigma'_2$.
- If $\langle \sigma_1, h \rangle \rightarrow_i \langle \sigma'_1, h' \rangle$ then exists σ'_2 such that $\sigma_2 \mapsto_i \sigma'_2$ and $\langle \sigma'_1, h' \rangle \mathcal{R} \sigma'_2$.

Finally we show that the security property of the TLS and the refinement relation directly transfer the mmu-integrity/no-exfiltration/no-infiltration to the implementation. We use Σ_I to represent the space of consistent implementation states: States σ_2 such that if $\langle \sigma_1, h \rangle \mathcal{R} \sigma_2$ then $I(\langle \sigma_1, h \rangle)$.

Corollary 1 (Implementation security transfer) *Let $\sigma_1, \sigma_2 \in \Sigma_I$, $i \in \{0, 1\}$, $O_g^r(\sigma_1) = O_g^r(\sigma_2)$:*

- if $\sigma_1 \mapsto_0 \sigma'_1$ then $\sigma_1 \equiv_{mmu} \sigma'_1$
- if $\sigma_1 \mapsto_0 \sigma'_1$ and $\neg S(\sigma_1)$ then $O_s^r(\sigma_1) = O_s^r(\sigma'_1)$
- if $\sigma_1 \mapsto_i \sigma'_1$, $\sigma_2 \mapsto_i \sigma'_2$, and $\neg S(\sigma_1)$ and $\neg S(\sigma_2)$ then $O_g^r(\sigma'_1) = O_g^r(\sigma'_2)$

6.3 Binary Code Correctness

In the ARMv7 model of Section 4 the untrusted guest, the trusted services and the hypervisor share the CPU, and the hypervisor behaviour is modelled by the execution of its binary instructions.

Intuitively, internal hypervisor states cannot be observed by the guest, since (i) during the execution of the handler the guest is not active, (ii) the hypervisor does not support preemption and (iii) the handlers do not raise nested exceptions. For this reason, we introduce a weak transition relation, which hides states that are privileged. We write $\sigma_0 \rightsquigarrow_i \sigma_n$ if there is a finite execution $\sigma_0 \rightarrow_i \dots \rightarrow \sigma_n$ such that $mode(\sigma_n) = PL0$ and $mode(\sigma_j) = PL1$ for $0 < j < n$.

Our goal is to exhibit a refinement property relating implementation states σ_1 and real states σ_2 . The refinement relation $\mathcal{R}'$ (that is left-total with the exception of the faulty state $\perp$ and surjective) requires that: (i) the registers and coprocessors contain the same value in both states, (ii) the guest memory contains the same values in both states, (iii) the memory holding the hypervisor data structures contains the same values in both states. For the observations of the guest O_g^r on real states, the same definition as for the implementation model are used, i.e. the guest can observe the same addresses in both models. Again the refinement is required to establish a bisimulation.

Theorem 7 (Real Refinement) *Let $\sigma_1, \sigma_2 \in \Sigma_I$ such that $\sigma_1 \mathcal{R}' \sigma_2$. Let $i \in \{0, 1\}$.*

- If $\sigma_2 \rightsquigarrow_i \sigma'_2$ then exists σ'_1 such that $\sigma_1 \rightarrow_i \sigma'_1$ and $\sigma'_1 \mathcal{R}' \sigma'_2$.
- If $\sigma_1 \mapsto_i \sigma'_1$ then exists σ'_2 such that $\sigma_2 \rightsquigarrow_i \sigma'_2$ and $\sigma'_1 \mathcal{R}' \sigma'_2$.

Finally one must show that the security properties are transferred to the real model.

Corollary 2 (Real Security Transfer) *Let $\sigma_1, \sigma_2 \in \Sigma_I$, $i \in \{0, 1\}$, $O_g^r(\sigma_1) = O_g^r(\sigma_2)$:*

- if $\sigma_1 \rightsquigarrow_0 \sigma'_1$ then $\sigma_1 \equiv_{mmu} \sigma'_1$
- if $\sigma_1 \rightsquigarrow_0 \sigma'_1$ and $\neg S(\sigma_1)$ then $O_s^r(\sigma_1) = O_s^r(\sigma'_1)$
- if $\sigma_1 \rightsquigarrow_i \sigma'_1$, $\sigma_2 \rightsquigarrow_i \sigma'_2$, and $\neg S(\sigma_1)$ and $\neg S(\sigma_2)$ then $O_g^r(\sigma'_1) = O_g^r(\sigma'_2)$

6.4 Execution Safety and End-to-End Information Flow Security

Note that we do not prove explicitly execution safety. The reason is that the transition relations of the ARM CPU and the TLS are left-total. Left-totality for the ARM CPU depends on the fact that the physical CPU never halts (with the exception of the privileged “wait” instruction that is never used by the hypervisor). Left-totality for the TLS holds because the virtualization API is modelled by HOL4 total functions over TLS states; every function is equipped with a termination proof, which is either automatically inferred by the theorem prover or has been manually verified. The only transitions that can yield a dead state ($\perp$) are the hypervisor transitions of the implementation model, due to incorrect memory accesses. Proving that this model can never reach the state $\perp$ is part of the proof of Theorem 6. It makes use of Lemma 9, which shows that all hypervisor memory accesses are correct.

We do not prove standard end-to-end information flow properties because their definitions depend on the actual trusted services. This is often the case when two components are providing services to each other. For example, if the trusted service is the run-time monitor of Section 12, then it should be able to directly read the memory of the untrusted Linux (to compute the signatures). Additionally, the trusted service can be allowed to affect the behaviour of the guest, for example by rebooting it or by changing its process table if a malware is detected.

However, our verification results enable the enforcement of end-to-end security by properly restricting the capability of the trusted services. In fact, these services are executed non-privileged, thus their execution is constrained by properties 1 and 2. Moreover, their memory mapping is static, is configured in the master page table of the hypervisor, and is independent of the guest configuration. If complete isolation is needed, it is sufficient to configure these entries of the master page table properly, use properties 1 and 2 together with Theorem 2 to prove that the trusted services cannot affect and are independent of the guest resources. This enables the trace properties to be established and consequently to obtain end-to-end security.

7 TLS Consistency

We proceed to describe the strategy for proving the TLS consistency properties of Section 6.1. To this end we summarise the structure of the system invariant. The system invariant consists of two predicates: one (RC) ensures soundness of the reference counter, and the other (TC) guarantees that the state of the system is well typed.

The reference counter is sound (i.e. $RC(\sigma, h)$) if for every physical block b , the reference counter $h.pgrefs(b)$ is equal to $\sum_{i \in \{0 \dots 2^{20} - 1\}} count(\langle \sigma, h \rangle, i, b)$, where $count$ is a function that counts the number of references from the block i to the block b , according to the reference-counter policy:

- if b is a data block and i is a page table, i.e. $h.pgtype(b) = D$ and $h.pgtype(i) \neq D$, then $count$ is the number of page table descriptors in i that are writable in non-privileged mode and that point to b ,
- if b is a L2 page table and i is a L1 page table then $count$ is the number of page table descriptors in i that use the table b to fragment the section, and

- if i is not a page table, i.e. $h.pgtype(i) = D$, then $count(\langle \sigma, h \rangle, i, b) = 0$.

A system state is well typed ($TC(\sigma, h)$), if the MMU is enabled, the current L1 page table is inside a physical block of type L1, and each physical block b that does not have type *data* ($h.pgtype(b) \neq D$) contains a sound page table ($sound(\langle \sigma, h \rangle, b)$) and resides in the guest memory ($pa \in G_m$ for all pa such that $pa[31 : 12] = b$). The predicate *sound* ensures that (i) no unpredictable setting is allowed, (ii) page tables grant write access only to blocks with type *data*, (iii) page tables forbid any access in *PL0* to blocks outside the guest memory, and (iv) every L1 page table descriptor points to a block typed L2. Section 5.2 and Figure 4 exemplify these constraints.

The proofs of the theorems of Section 6.1 have been obtained using the HOL4 theorem prover and the lemmas are described in the following.

Lemma 1 (Invariant implies MMU-safety) *If $\langle \sigma, h \rangle \in \mathcal{Q}_I$ then $mmu_s(\sigma)$.*

Lemma 1 demonstrates an important property of the system invariant; a state that satisfies the invariant has the same MMU behaviour as any state whose memory differs only for addresses that are writable in unprivileged mode. The proof of the lemma depends on the formal model of the ARMv7 MMU (but not on its instruction set); there the MMU behaviour is determined by coprocessor registers and the contents of the active L1 and referenced L2 page tables. The invariant guarantees that the active L1 page table of σ resides in four consecutive blocks that have type L1 and every page table descriptor in this table points to a block typed L2. Moreover, only *data* blocks may be writable in unprivileged mode and write attempts to other blocks will be rejected. We examine a state σ' that is write-derivable in unprivileged mode from σ , but has the same coprocessor registers, selecting the same active L1 page table. Since the content of the page tables is unchanged, the MMU in σ' behaves exactly like in σ .

Proof sketch of Theorem 2 To demonstrate MMU-integrity the ARM-integrity property is used. By definition of the TLS transition relation, if $\langle \sigma, h \rangle \rightarrow_0 \langle \sigma', h' \rangle$ then (in the ARM model) $\sigma \rightarrow_{PL0} \sigma'$. Moreover, Lemma 1 guarantees $mmu_s(\sigma)$. Thus, Property 1 can be used to conclude that $wd(\sigma, \sigma', PL0)$ and $\sigma.coregs = \sigma'.coregs$, i.e. σ' is a state write-derivable from σ and coprocessor registers have not changed. Finally, it suffices to apply Definition 4 (MMU-safety) to show that $\sigma \equiv_{mmu} \sigma'$.

Lemma 2 (Guest isolation) *Let $\langle \sigma, h \rangle \in \mathcal{Q}_I$. For every physical address pa and access request req if $\neg S(\sigma)$ and $mmu_{ph}(\sigma, PL0, pa, req)$ then $G_m(pa)$.*

The proof of Lemma 2 uses the formal ARMv7 MMU model and directly follows from the invariant. In particular, part (iii) of predicate *sound* demands that entries of a page table grant access permissions to the guest only if the entry points to a physical address that is inside the guest memory.

Proof sketch of Theorem 4 Similar to the proof of Theorem 2, the definition of the transition relation and Lemma 1 yield that $\langle \sigma, h \rangle \rightarrow_0 \langle \sigma', h' \rangle$ implies $h = h'$, $\sigma \rightarrow_{PL0} \sigma'$ and $mmu_s(\sigma)$. Then Property 1 gives $\sigma.coregs = \sigma'.coregs$ and $wd(\sigma, \sigma', PL0)$, meaning that (according to the contraposition of Definition 2) the memories of σ and σ'

contain the same value for every physical address that is not writable in mode $PL0$ in σ . By Lemma 2 the guest can only obtain write permissions to the physical addresses belonging to its own memory, thus the memories of σ and σ' have the same value for every physical address not in G_m . Moreover banked registers cannot be changed in unprivileged mode. Consequently, $O_s(\langle\sigma, h\rangle) = O_s(\langle\sigma', h'\rangle)$ holds as claimed.

Proof sketch of Theorem 5 We proceed separately for unprivileged and privileged transitions. For unprivileged transition the ARM-confidentiality property is used. As proven above, from the definition of the transition relation, Lemma 1, $\langle\sigma_1, h_1\rangle \rightarrow_0 \langle\sigma'_1, h'_1\rangle$, and $\langle\sigma_2, h_2\rangle \rightarrow_0 \langle\sigma'_2, h'_2\rangle$ we obtain $h_1 = h'_1$, $h_2 = h'_2$, $\sigma_1 \rightarrow_{PL0} \sigma'_1$, $\sigma_2 \rightarrow_{PL0} \sigma'_2$, $mmu_s(\sigma_1)$ and $mmu_s(\sigma_2)$. Since $O_g(\langle\sigma_1, h_1\rangle) = O_g(\langle\sigma_2, h_2\rangle)$, the user registers, guest memories (i.e. the content for addresses in G_m), and co-processor registers are the same in σ_1 and σ_2 . The definition of $mmu_s(\sigma_1)$ yields $\sigma_1 \equiv_{mmu} \sigma_2$. Moreover, Lemma 2 shows that the guest can obtain an access permission only to the physical addresses in G_m , thus the memories of σ_1 and σ_2 contain the same value for every address in $G_m \supseteq \{pa \mid \exists req. mmu_{ph}(\sigma_1, PL0, pa, req)\}$. This enables Property 2, which in turn justifies that $\sigma'_1.uregs = \sigma'_2.uregs$ and $\forall pa \in G_m. \sigma'_1.mem(pa) = \sigma'_2.mem(pa)$, i.e. the guest observations in σ'_1 and σ'_2 are the same.

The proof of the Theorem 5 for hypervisor transitions has been obtained by performing relational analysis. The function H_a accesses only three state components: the hypervisor data structures (i.e. h), the user registers and the memory (in order to validate page tables). The function H_a is symbolically evaluated on the states $\langle\sigma_1, h_1\rangle$ and $\langle\sigma_2, h_2\rangle$; whenever H_a updates an intermediate variable, it must be demonstrated that the value of the variable is the same in both executions, whenever H_a modifies a state component (e.g. memory or register), it must be demonstrated that the equivalence of guest observation is preserved. These tasks are completely automatic for assignments that only depend on intermediate variables and user registers. For every assignment that depends on memory accesses, a new verification condition is generated to require that the accessed addresses are the same in both executions and to guarantee that such address is in the guest memory. Finally, these verification conditions are verified, showing that H_a never accesses memory outside G_m .

Finally, we prove Theorem 1 by showing that the invariant is preserved first by guest transitions (Lemma 3) and then by the abstract handlers (Lemma 4).

Lemma 3 (Invariant vs guest) *Let $\langle\sigma, h\rangle \in \mathcal{Q}_I$. If $\langle\sigma, h\rangle \rightarrow_0 \langle\sigma', h'\rangle$ then $I(\langle\sigma', h'\rangle)$.*

This lemma demonstrates that the invariant is preserved by guest transitions. Its proof depends on the ARM-integrity property. It is straightforward to show that the invariant only depends on the content of the physical blocks that are not typed D and the hypervisor data (i.e. h and h'). Similar to the proof of Theorem 4, the definition of the transition relation, Lemma 1 and Property 1 guarantee that if $\langle\sigma, h\rangle \rightarrow_0 \langle\sigma', h'\rangle$ then $h = h'$, $\sigma \rightarrow_{PL0} \sigma'$, $mmu_s(\sigma)$ and $wd(\sigma, \sigma', PL0)$. As in the proof of Lemma 1 it is shown that in σ every block that is not typed D is not writable, concluding that the invariant is preserved.

Lemma 4 (Invariant vs hypervisor) *Let $\langle \sigma, h \rangle \in \mathcal{Q}_I$. If $\langle \sigma, h \rangle \rightarrow_1 \langle \sigma', h' \rangle$ then $I(\langle \sigma', h' \rangle)$.*

The lemma demonstrates that the invariant is preserved by the handler functionalities and shows the functional correctness of the TLS design. By definition, if $\langle \sigma, h \rangle \rightarrow_1 \langle \sigma', h' \rangle$ then there exists σ'' such that $\sigma \rightarrow \sigma''$, $\text{mode}(\sigma'') = \text{PL1}$ and $\langle \sigma', h' \rangle = H_a(\langle \sigma'', h \rangle)$. Similar to the proof of Lemma 3, Property 1 is used to guarantee that the invariant is preserved by this transition: $I(\langle \sigma'', h \rangle)$. Then we show that the invariant is preserved by the abstract handler H_a .

This verification task requires the introduction of several supporting lemmas. The idea is that according to the input request, the abstract handler only changes a small part of the system state. For instance, when H_a maps a section, only the current L1 page table is modified, the contents of other blocks are unchanged. In order to demonstrate that the invariant is indeed preserved for the parts of the state that are not affected by H_a , we introduce a number of additional lemmas. These lemmas are sufficiently general to be used to verify different virtualization mechanisms that involve direct paging and they prove the intuition that the type of a block can be safely changed when its reference counter is zero.

Definition 6 *Let h and h' be two abstract hypervisor states. The predicate $\text{type}_s(h, h')$ holds if and only if $h.\text{pgtype}(b) \neq h'.\text{pgtype}(b)$ implies $h.\text{refs}(b) = 0$ for all blocks b .*

Changing the type of a block can affect the soundness of page tables that reference that block. The following lemma expresses the key property that soundness of page tables is preserved for all type changes of other blocks, as long as the reference counters of that blocks are zero:

Lemma 5 *Assume $I\langle \sigma, h \rangle$ and $\text{type}_s(h, h')$. For every block b such that $h.\text{pgtype}(b) = h'.\text{pgtype}(b)$, if $\text{sound}(\langle \sigma, h \rangle, b)$ then $\text{sound}(\langle \sigma, h' \rangle, b)$.*

The proof of Lemma 5 hinges on the fact that type changes can only break parts (ii) and (iv) of the page table soundness condition. However, if the type is only changed for blocks that are not referenced by any page table, soundness is preserved trivially.

We exemplify the usage of Lemma 5 when proving Lemma 4. Assume that H_a is allocating a new L2 page table in the block b' (i.e. changing the type of b' from D to $L2$). This operation can break soundness of any other block b . In fact, b can be a page table containing a writable mapping to b' , thus b is sound in $\langle \sigma, h \rangle$ but is unsound in $\langle \sigma, h' \rangle$. The side condition $\text{type}_s(h, h')$ ensures that this case cannot occur: to safely allocate a new page table, the reference counter of b' must be zero, thus b cannot contain a writable mapping to b' .

Similarly, the following lemma shows that, if the page type is changed only for blocks with zero references, then for all other page tables, the number of references is unchanged.

Lemma 6 *Assume $I\langle \sigma, h \rangle$ and $\text{type}_s(h, h')$. For all blocks b, b' if $h.\text{pgtype}(b) = h'.\text{pgtype}(b)$ then $\text{count}(\langle \sigma, h \rangle, b, b') = \text{count}(\langle \sigma, h' \rangle, b, b')$.*

Finally we use the following lemma to show that the well-typedness of a block and its counted outgoing references are independent from the content of the other physical blocks.

Lemma 7 *Let $\sigma, \sigma' \in \Sigma$ such that $I\langle\sigma, h\rangle$. If σ and σ' have the same memory content for the block b then $\text{sound}(\langle\sigma', h\rangle, b)$ and for every block b' $\text{count}(\langle\sigma', h\rangle, b, b') = \text{count}(\langle\sigma, h\rangle, b, b')$.*

For every functionality of the virtualization API (see Figure 3), Lemmas 5, 6 and 7 help to limit the proof of Lemma 4 to only checking the well-typedness and soundness of the reference counter for the blocks that are affected by H_a .

Proof of Theorem 1 The theorem directly follows from Lemmas 3 and 4.

8 Refinement

To verify the implementation refinement relation (i.e. prove Theorem 6) we proved two auxiliary lemmas:

Lemma 8 (Real MMU) *Let $\langle\sigma_1, h\rangle \in \mathcal{Q}_I$ and $\sigma_2 \in \Sigma$. If $\langle\sigma_1, h\rangle \mathcal{R} \sigma_2$ then $\sigma_1 \equiv_{mmu} \sigma_2$.*

The Lemma shows that TLS and implementation states have the same MMU configuration. Its proof uses that the system invariant requires page tables to be allocated inside the guest memory, whose content is the same in the TLS and implementation states. Moreover, coprocessor registers contain the same data.

Lemma 9 (Hypervisor page tables) *Let $\langle\sigma_1, h\rangle \in \mathcal{Q}_I$ and $\sigma_2 \in \Sigma$. If $\langle\sigma_1, h\rangle \mathcal{R} \sigma_2$ then:*

1. *For all pa and req , if $pa \in G_m$ then $mmu(\sigma_2, Gpa2va(pa), PL1, req) = pa$.*
2. *For every block b and access request req , $mmu(\sigma_2, tbl_{va} + 4 * b, PL1, req) = tbl_{pa} + 4 * b$, where tbl_{pa} is the physical address where the hypervisor data structure is allocated.*

The Lemma shows that the implementation is always able to access the guest memory and the hypervisor data structures, and that the computed physical addresses match the expected values.

Proof sketch of Theorem 6 To prove that the refinement is preserved by all possible transitions we verify independently the guest and hypervisor transitions. For guest transitions, Theorem 4 (No-Exfiltration) and Lemma 1 (MMU-safety) guarantee that the guest can change neither the memory outside G_m nor the page tables. Thus it is sufficient to show that the physical addresses of the hypervisor data structures are outside the guest memory. Moreover, Theorem 5 (No-Infiltration) guarantees that the guest transition is not affected by a part of the state that is not equivalent in $\langle\sigma_1, h\rangle$ and σ_2 . For the hypervisor transition we used a compositional approach. First, we verified that all low-level operations (i.e. reads and updates of the page tables, reads and updates of the hypervisor data structures) preserve the refinement relation. Then we compose these results to show that the TLS and implementation transitions behave equivalently.

Proof sketch of Corollary 1 The proof depends on the fact the the relation $\mathcal{R}$ is left-total and surjective. Proving that the security properties of the TLS are transferred to the implementation model is simplified by the definition of the refinement relation. For example, Lemma 8 and Theorem 2 are used to show that the MMU configuration cannot be changed by the untrusted guest. Assume $\sigma_2 \mapsto_0 \sigma'_2$ and let $\langle \sigma_1, h \rangle$ be a TLS state such that $\langle \sigma_1, h \rangle \rightarrow_0 \langle \sigma'_1, h' \rangle$ and $\langle \sigma_1, h \rangle \mathcal{R} \sigma_2$. Since the refinement is preserved by all transitions (Theorem 6), exists $\langle \sigma'_1, h' \rangle$ such that $\langle \sigma'_1, h' \rangle \mathcal{R} \sigma'_2$. Lemma 8 yields $\sigma_1 \equiv_{mmu} \sigma_2$ and Theorem 2 (MMU-integrity) guarantees that $\sigma_1 \equiv_{mmu} \sigma'_1$, thus $\sigma_2 \equiv_{mmu} \sigma'_1$. Finally, Lemma 8 yields $\sigma'_1 \equiv_{mmu} \sigma'_2$, thus $\sigma_2 \equiv_{mmu} \sigma'_2$. Similar reasoning is used to prove that properties *no-exfiltration* and *no-infiltration* are transferred to the implementation model, by showing that, if two TLS states have the same observations (i.e. $O_g(\langle \sigma_1, h \rangle) = O_g(\langle \sigma'_1, h' \rangle)$ or $O_s(\langle \sigma_1, h \rangle) = O_s(\langle \sigma'_1, h' \rangle)$) and the states are refined by two implementation states (i.e. $\langle \sigma_1, h \rangle \mathcal{R} \sigma_2$ and $\langle \sigma'_1, h' \rangle \mathcal{R} \sigma'_2$), then the two implementation states have the same observations (i.e. $O_g(\sigma_2) = O_g(\sigma'_2)$ or $O_s(\sigma_2) = O_s(\sigma'_2)$).

9 Binary verification

Binary analysis is key requirement to ensure security of low-level software platform, like hypervisors. Machine code verification obviates the necessity of trusting the compilers. Moreover, low level programs mix structured code (e.g. implemented in C) with assembly and use instructions (e.g. mode switches and coprocessor interactions) that are not part of the high level language, thus making difficult to use verification tools that target user level code.

For our hypervisor the main goal of the verification of the binary code is to prove Theorem 7. This verification relies on Hoare logic and requires several steps. The first step (Section 9.2) is transforming the relational reasoning into a set of contracts for the hypervisor handlers and guaranteeing that the refinement is established if all contracts are satisfied. Let C be the binary code of one of the handlers, the contract $\{P\}C\{Q\}$ states that if the precondition P holds in the starting state of C , then the postcondition Q is guaranteed by C .

Then, we adopt a standard semi-automatic strategy to verify the contracts. First, the weakest liberal precondition $WLP(C, Q)$ is computed on the starting state, then it is verified that the precondition P implies the weakest precondition.

The second verification step (computation of weakest preconditions) can be performed directly in HOL4 using the ARMv7 model. However, this task requires a significant engineering effort. We adopted a more practical approach, by using the Binary Analysis Platform (BAP) [9]. The BAP tool-set provides platform-independent utilities to extract control flow graphs and program dependence graphs, to perform symbolic execution and to perform weakest-precondition calculations. These utilities reason on the BAP Intermediate Language (BIL), a small and formally specified language that models instruction evaluation as compositions of variable reads and writes in a functional style.

The existing BAP front-end to translate ARM programs to BIL lacks several features required to handle our binary code: Support of ARMv7, management of proces-

sor status registers, banked registers for privileged modes and coprocessor registers. For this reason we developed a new front-end, which is presented in Section 9.3, that converts an ARMv7 assembly fragment to a BIL program.

The final verification step consists of checking that the precondition P implies the weakest precondition. This task can be fully automated if the predicate $P \Rightarrow WLP(C, Q)$ is equivalent to a predicate of the form $\forall \vec{x}. A$ where A is quantifier free. The validity of A can then be checked using a Satisfiability Modulo Theory (SMT) solver that supports bitvectors to handle operations on words. In this work, we used STP [23].

An alternative approach for binary verification is to use the “decompilation” procedure developed by Myreen [39]. This procedure takes an ARMv7 binary and produces a HOL4 function that behaves equivalently (i.e. implements the same state transformation). This result allows to lift the verification of properties of assembly programs to reasoning on HOL4 functions. However, the latter task can be expensive due to the lack of automation in HOL4.

9.1 Soundness of the verification approach

The procedure described here to establish the functional correctness of the hypervisor code relies on four main arguments.

1. The HOL4 procedures that evaluate the effects of a given instruction in the ARMv7 model specify the updates to the processor state correctly. We use the ARMv7 step theorems to guarantee the correctness of this task.
2. The lifter transforms this state update information into an equivalent list of single-variable assignments in BIL. The correctness of this part of the lifter is an assumption for now.
3. The expressions in each update of a processor component are correctly translated to BIL expressions in the list of assignments, preserving their semantics. This has been proven for our lifter.
4. The binary code fragment that is lifted is actually executed on the ARMv7 hardware.

The last argument relies on the fact that the boot loader places the unmodified hypervisor image to the right place in memory. This is another assumption since we do not verify our boot loader. Furthermore there must not be self-modifying code. The easiest way to enforce this is to partition the hypervisor memory via its page table into data and code region and prove an invariant that the first is non-executable but the latter is non-writeable. There is no such protection against self-modifying code in the hypervisor at the moment. Finally, one needs to show that the binary code is not interrupted, thus proving that the hypervisor is in deed non-preemptive. We do not have a full proof of the statement, but there are provisions in the lifter to show the absence of ARMv7 interrupts and exceptions.

For system call and unknown instructions, the lifter generates BIL statements that always fail, i.e., one can only verify programs in BAP that do not use such instructions.

We follow the same approach for fetches, jumps, and memory instructions accessing constant addresses which are not mapped in the hypervisor’s static page table. Thus such operations cannot produce pre-fetch or data aborts. Additional care has to be taken to distinguish data and code regions to avoid permission faults due to writes to the code region or fetches from the data region, however there are no such checks at the moment. Indirect jumps are solved dynamically based on the lifted BIL program (see Sect. 9.4) and for any jump to a location that is not defined in the program, i.e., not in the region accessible by the hypervisor, analysis with BAP will give an error. For dynamic memory accesses the lifter is able to insert assertions that the corresponding address is mapped, however the feature is currently not activated automatically. At last, the reception of external interrupts should not be re-enabled during hypervisor execution. Currently this invariant is not checked in the code verification but it could be easily added as an assertion between every instruction.

9.2 Generation of the contracts

Let C be the binary code of one of the handlers, we define the precondition P and the postcondition Q such that the contract subsumes the refinement:

- $P(\sigma_2) = \text{exists } \sigma_1 \text{ such that } \sigma_1 \mathcal{R}' \sigma_2$
- $Q(\sigma'_2, \sigma_2) = \text{for all } \sigma_1, \sigma'_1 \text{ if } \sigma_1 \mathcal{R}' \sigma_2 \text{ and } \sigma_1 \mapsto_1 \sigma'_1 \text{ then } \sigma'_1 \mathcal{R}' \sigma'_2$

These contracts are not directly suitable for the verification of the binary code because the contracts quantify on states (σ_1 and σ'_1) that are in relation with the pre-state (σ_2) and post-state (σ'_2) of the binary code. We developed an HOL4 inference procedure specific for the structure of our hypervisor. The output of the procedure is a proof guaranteeing that the original contract $\{P\}C\{Q\}$ is satisfied if a “simplified” contract $\{P'\}C\{Q'\}$ is met. That is, for every σ_2, σ'_2 the predicate $P'(\sigma_2) \Rightarrow Q'(\sigma'_2, \sigma_2)$ implies $P(\sigma_2) \Rightarrow Q(\sigma'_2, \sigma_2)$.

This procedure makes heavy use of the simplification rules and decision procedures of HOL4. We informally summarise how this procedure works for the memory resource. The precondition P' is generated by transferring the hypervisor invariant I from the abstract model down to the real model. This is possible because (i) $\mathcal{R}'$ constrains the memory holding the hypervisor data structures to be the same in σ_2 and σ_1 , (ii) $\mathcal{R}$ (the refinement between the abstract model and the implementation model) guarantees that this memory area contains a projection of the hypervisor data structures in the TLS state, (iii) on the TLS state the hypervisor invariant holds.

For the postcondition Q' we proceed as follows. If $\sigma_1 \rightarrow_1 \sigma'_1$ then $\sigma'_1 = H_r(\sigma_1)$. Let A be the set of memory addresses that are constrained by the refinement relation $\mathcal{R}'$ and let B be the set of addresses that are modified by H_r . The set B is usually easy to identify in HOL4, thanks to its symbolic execution capability and the lemmas that have been already proven for the tasks of Section 8. For each handler we demonstrate that $B \subseteq A$.

For every address $a \in (\bar{B}) \cap A$ (namely addresses constrained by the refinement relation that are not updated) we add to Q' the constraint $\sigma'_2.\text{mem}(a) = \sigma_2.\text{mem}(a)$.

This uses $\sigma_1.mem(a) = \sigma_2.mem(a)$ and the refinement for the address a ($\sigma'_1.mem(a) = \sigma'_2.mem(a)$).

For every address $a \in B$ we make use of the HOL4 rewriting engine to obtain a naive symbolic execution of the handler specification. We use HOL4 to symbolically compute $H_r(\sigma_1)$ then we use the precondition $\sigma_1 \mathcal{R}' \sigma_2$ to rewrite the result and make sure that this is expressed only in terms of σ_2 . Let exp be the resulting expression, we add to Q' the constraint $\sigma'_2.mem(a) = exp.mem(a)$.

When the symbolic execution is too complex (e.g. too many outcomes are possible according to the initial state), we split the verification by generating multiple contracts $\{P_1\}C\{Q_1\}, \dots, \{P_n\}C\{Q_n\}$, where $P_i = P(\sigma_2) \wedge A_i(\sigma_2)$ and $\bigvee_i A_i$ is a valid formula (i.e. all possible cases are taken into account).

9.3 Translation of ARMv7 to BIL

The target language of the ARMv7 BAP front-end is BIL, a simple single-variable assignment language tailored to model the behaviour of assembly programs and developed to be platform independent. A BIL program is a sequence of statements. Each statement can affect the system state by assigning the evaluation of an expression to a variable, (conditionally or unconditionally) modifying the control flow, terminating the system in a failure state if an assertion does not hold and unconditionally halting the system in a successful state. The BIL data types for expressions and variables include boolean, words of several sizes and memories. The main constituent of BIL statements are expressions, that include constants, standard bit-vector binary and unary operators, and type casting function. Additionally, an expression can read several words from a memory or generate a new memory by changing a word in a given one.

We developed the new front-end on top of the HOL4 ARM model (see Section 4), so that the soundness of the transformation from an ARM assembly instruction to its corresponding BIL program relies on the correctness of the ARM model used in HOL4 and not on a possibly different formalization of ARMv7. Our approach is illustrated in Figure 7.

The HOL4 model provides a mechanism to statically compute the effects of an instruction via the *arm_steps* function. Let *inst* be the encoding of an instruction, then *arm_steps(inst)* returns the possible execution steps $\{st_1, \dots, st_n\}$. Each step $st_i = (c_i, t_i)$ consists of the condition c_i that enables the transition and the function t_i that transforms the starting state into the next state. The function *arm_steps* is a certifying HOL4 procedure, since its output is a theorem demonstrating that for every $\sigma \in \Sigma$ if *fetch*(σ) = *inst* and $c_i(\sigma)$ holds then $\sigma \rightarrow_{PL1} t_i(\sigma)$. For standard ARM decoding the function *fetch* reads four bytes from memory starting from the address pointed to by the program counter.

The translation procedure involves the following steps, (i) mapping HOL4 ARM states to BIL states and (ii) for each instruction of the given assembly fragment producing the BIL fragment that emulates the *arm_steps* outputs. To map an ARM state to the corresponding BIL state we use a straightforward approach. A BIL variable is used to represent a single component of the machine state: for example, the variable *R0* represents the register number zero and the variable *MEM* represents the system memory.

Figure 7: Lifting machine code to BIL using the HOL4 ARMv7 model. The *arm_steps* function translates machine instructions into steps st_i consisting of guards c_i and transition functions t_i . Their effect is equivalent to the hypervisor computation in the real model (states 3-7, cf. Fig. 1). Each step st_i is in turn translated into equivalent BIL code.

To transform an ARM instruction to the corresponding BIL fragment we need to capture all the possible effects of its execution in terms of affected registers, flags and memory locations. The generated BIL fragment should simulate the behaviour of the instruction executed on an ARM machine. Therefore, to obtain a BIL fragment for an instruction we need to translate the predicates c_i and their corresponding transformation functions t_i . This task is accomplished using symbolic evaluation of the predicates and the transformation functions. The input of the evaluation is a symbolic state in which independent variables are used to represent each state register, flag, coprocessor register and memory. This approach allows us to obtain a one-to-one mapping between the symbolic state variables and the BIL state variables. To transform a predicate c_i , we apply the predicate to a symbolic ARMv7 state, thus obtaining a symbolic boolean expression in which free-variables are a subset of the symbolic state variables. Similarly, to map a transformation function t_i , we apply t_i to a symbolic state, thus obtaining a new state in which each register, flag and affected memory location is assigned a symbolic expression. Intuitively, for each instruction we produce the following BIL fragment:

```
label GUARD_1
cjmp  $|b_1|$ , EFFECT_1, GUARD_2
...
label GUARD_N
cjmp  $|b_n|$ , EFFECT_n, ERROR
label ERROR
assert false
```

Where “*cjmp*” is the BIL instruction for conditional jump and $|b_i|$ is a BIL boolean expression obtained by translating the symbolic evaluation of c_i . Then, for each step i we

symbolically evaluate the transformation t_i and for each field (i.e. memory locations, registers, flags and coprocessor registers) that has been updated we transform the resulting symbolic expression and assign it to the corresponding BIL variable, generating a fragment

```
label EFFECT_i
var_1 = |exp_1|
...
var_n = |exp_n|
```

The described lifting procedure is straightforward. However, its soundness depends on the correct transformation of HOL4 terms (e.g. $|b_n|$ and $|exp_n|$) to BIL expressions. Since the number of HOL4 operators that occur in the generated expressions is huge, we cannot rely on a simple syntactical transformation to obtain a robust conversion of them to BIL. Moreover, the transformation of HOL4 terms to BIL expressions is used to convert the pre/post conditions of our contracts from HOL4 to BAP. For this reason, we formally modelled in HOL4 the BIL expression language (by providing a deep embedding of BIL expression in HOL4) and the translation procedure *liftExp* certifies its output:

$$\text{liftExp}(exp) = (exp', \vdash exp = exp')$$

In particular, the translation procedure yields a theorem demonstrating that the HOL4 input term exp is equivalent to the BIL expression exp' .

In order to dynamically generate the certifying theorem, the translation procedure is implemented in ML, which is the HOL4 meta language. The translation syntactically analyses and deconstructs the input expressions to select the theorems to use in the HOL4 conversion and rewrite rules. For terms composed by nested expressions the procedure acts recursively.

9.4 Supporting tools

To compute the weakest precondition of a program is necessarily to statically know the control flow graph (CFG) of the program. This means that the algorithm depends on the absence of indirect jumps. Even if the hypervisor C-code avoids their explicit usage (e.g. by not using function pointers), the compiler introduces an indirect jump for each function exit point (e.g. the instruction at the address $0x20C$ in Figure 8, is an indirect jump). Solving an indirect jump (i.e. enumerating all possible locations that can be target of the jump) is depending on checking the correctness of other properties of the application (e.g. the link register, which is usually used to track the return address of functions, can be pushed and popped from the stack, thus making the correctness of the control flow dependent on the integrity of the stack itself). Since we are interested in solving indirect jumps of code fragments that must respect contracts (Hoare triples $\{P\}C\{Q\}$), we implemented a simple iterative procedure that uses STP to discover all possible indirect jump targets under the contract precondition P .

1. The CFG of the of C fragment is computed using BAP. From the CFG, the list L of reachable addresses containing an indirect jump is extracted
2. For each address $a \in L$, the code fragment C is modified as follows:

- (a) let exp_a be the expression used in the indirect jump
- (b) the indirect jump is substituted with an assertion, which requires exp_a to be different from a fresh variable fv_a ; if such assertion fails, i.e. $exp_a = fv_a$, the modified fragment C terminates with a fault, otherwise it correctly terminates
3. the new fragment has no indirect jump; the weakest precondition WP of the postcondition $true$ (i.e. correct termination) is computed
4. the SMT solver searches for an assignment of the free variables (including all fv_i) that invalidates $P \Rightarrow WP$
5. if the SMT solver discovers a counterexample which involves the indirect jump at the address a , then it also discovers a possible target for this jump via selected assignment of the variable fv . Let exp be the expression used in the indirect jump. The fragment C is transformed by substituting the indirect jump with a conditional statement; if exp is equal to fv then jump to the fixed address fv , otherwise jump to the expression exp : `jmp exp` will be transed into

```

cjmp exp == fv; value; new_label
label new_label: jmp exp

```
6. if the SMT solver does not find a counterexample, then every indirect jump is either unreachable or all its possible targets have been discovered. The fragment C is transformed by substituting every indirect jump with an assertion that always fails (`assert false`).
7. The procedure is restarted. Note that the inserted conditional statements prevent that the discovered assignments of fv_a can be used to invalidate the formula by the SMT solver in the next iteration.

In order to handle the greater complexity of the hypervisor code respect to the separation kernel verified in [19], we re-engineered this tool as a BAP plug-in. A particular problem that we face is that the CFG can contain loops if the same internal function of the hypervisor is called twice from different points in the program. Integrating the procedure with BAP allowed us to reuse the existing loop-unfolding algorithms to break these artificial loops.

We use figures 8 and 9 to demonstrate the algorithm. The assembly program (Figure 8a) contains a function at 0×200 , which is invoked twice (from 0×100 and 0×108). This function push the link register in the stack (0×200), writes the content of the register R2 into the memory pointed by R1 (0×204), pop the link register in the stack (0×208) and returns ($0 \times 20C$). We assume that the precondition used is strong enough to ensure correct manipulation of the stack (e.g. the value of the stack pointer SP and the value of register R1 used as pointer in the instruction at 0×204 are distant at least one word). Figure 8b and 9a depict the BIL translation of the program and its initial CFG respectively. The CFG has only one reachable indirect jump (in $0 \times 20C$), whose expression is LR. The SMT solver discovers a possible target for this jump (in this case 0×104) and the program is transformed by substituting the indirect

0x100 b1 #0x200	label pc_0x100 PC = 0x100;
	LR = PC+4; jmp pc_0x100
0x104 ...	label pc_0x104 ...
0x108 b1 #0x200	label pc_0x108 PC = 0x108;
	LR = PC+4; jmp pc_0x100
0x10C ...	label pc_0x10C ...
// function	
0x200 push LR	label pc_0x200 PC = 0x200;
	mem=store32(mem, SP, LR);
	SP=SP-4
0x204 str R1, R2	label pc_0x204 PC = 0x204;
	mem=store32(mem, R1, R2);
0x208 pop LR	label pc_0x208 PC = 0x208;
	LR=load32(mem, SP+4);
	SP=SP+4
0x20C b LR	label pc_0x20C PC = 0x20C;
...	jmp LR

(a) Assembly

(b) BIL

Figure 8: Indirect jump example

Figure 9: Execution of the indirect jump solver

jump with a conditional statement, obtaining CFG is depicted in Figure 9b. This CFG has an artificial loop due to the two invocations of the same function. Figure 9c depicts the CFG obtained by unrolling the loop once. The program has now two reachable indirect jumps, the procedure is repeated and the SMT solver discovers that $0 \times 10C$ is a possible target of the jump in $0 \times 20C-1$. The CFG is transformed as Figure 9d. This CFG has still two indirect jumps. However, the SMT solver discovers that there is no assignment to the initial variables of the program that enables the activation of these jumps. Thus all indirect jumps have been resolved, the remaining ones are unreachable and are suppressed, obtaining the CFG in Figure 9e.

In addition to solving indirect jumps, effective application of the verification strategy required the implementation of several tools and optimisation of the weakest precondition algorithm of BAP. Weakest preconditions can grow exponentially with regard to the number of instructions. Even though this problem cannot be solved in general, we can handle the most common case for ARM binaries, namely the sequential composition of several conditionally executed arithmetical instructions. This pattern matches the optimisation performed by the compiler to avoid small branches. We improved the BAP weakest precondition algorithm by adding a simplification function that identifies these cases. For some fragments of the code this straightforward strategy strongly reduced the size of the precondition; e.g. for one fragment consisting of 27 C lines compiled to 35 machine instructions the size of the precondition has been reduced from 8 GB to 15 MB.

Furthermore, machine code (and BIL) lacks information on data types (except for the native types like word and bytes) and represents the whole memory as a single array of bytes. Writing predicates and invariants is complex because their definition depends on location, alignment and size of data-structure fields. Moreover, the behaviour of compiled code often depends on the content of static memory used to represent constant values of the high level language. We developed a set of tools that integrate HOL4 and GDB to extract information from the C source code and the compiled assembly. With the support of these tools we are able to write the invariants and contracts of the hypervisor independently of the actual symbol locations and data structure offsets produced by the compiler.

Figure 10 summarises the work-flow of our binary verification approach.

9.5 Limitations

The binary verification of the hypervisor has not been completed yet due to some time consuming tasks that require better automation. First, the inference procedure of Section 9.2 uses the HOL4 simplification rules and decision procedures, however it is not completely automatic and must be adapted for every handler. Without taking into account the specificity of each handler, a naive procedure can easily generate contracts that cannot be handled by SMT solvers. For every handler, we manually specialize the procedure to generate contracts that have no quantifier in the precondition and only universal quantifiers in the postcondition.

Further complexity arises due to presence of loops. In theory, loops can be automatically handled by unfolding, since all loops in the hypervisor code iterate over fixed and limited ranges (e.g. the number of descriptors in a page table). Practically,

be violated by either the guest or the trusted services. Moreover, the untrusted guest cannot directly affect the trusted services or directly extract information from their states (Theorems 4 and 5). This isolation is achieved by the complete mediation of the MMU settings and the allocation of the ARM domains 2-15 to the secure services. This approach limits the number of secure services to fourteen. However, this mechanism has the benefit of using the same page tables for both the guest and the trusted services (by reserving an area of the hypervisor virtual memory for the latter). This reduces the cost of context switch, since TLB and caches do not need to be cleaned. If more trusted services are needed, a separate page tables can be used.

The core of the hypervisor is the virtualization of the memory subsystem. This is provided by the handlers that are the subject of the verification and that are modelled by the transformations H_a and H_r (Section 6). This core have been extended with additional handlers to provide further functionalities, which are needed to host a complete OS and to implement useful secure services. Since these additional handlers are not involved in the virtualization of the memory subsystem, establishing that they preserve the invariant (Theorem 1) usually requires only to demonstrate that they do not directly change the physical blocks that contain the page tables and their memory safety.

10.1 Linux Support

The Linux kernel 2.6.34 has been modified to run on top of the hypervisor. This task required modification of architecture-dependent parts of the Linux kernel like execution modes, low-level exception routines and page table management. High-level OS functions such as process, resource and memory manager, file system, and networking did not require any modifications. This also introduce the additional handlers of the hypervisor that are not part of the formal verification.

CPU Privilege Modes In the absence of hardware supports, like virtualization extension, the target CPU includes only two execution modes: privileged and unprivileged (user). As for other approaches based on paravirtualization, since the hypervisor executes as privileged, the Linux kernel has been modified to execute as unprivileged. To separate kernel and user applications, the hypervisor manages two separate unprivileged execution contexts: virtual user and virtual kernel modes. In x86 these virtual modes can be implemented by segmentation. This approach is not possible for CPUs that do not provide this feature (e.g. x86 64-bit and ARM). Instead, we reserve the ARM domain 0 for the kernel virtual mode. Whenever the guest kernel requests a switch to virtual user mode (invoking the dedicated hypercall) we disable the domain 0, thus any access to the kernel virtual addresses generates a fault.

Note that the main security goal here is not to guarantee this OS-internal isolation, but to maintain the separation between the virtualized components (such as the Linux guest vs. secure data or services residing in non-guest memory).

CPU Exceptions CPU exceptions such as aborts and interrupts change the processor mode to privileged. These exceptions must therefore be handled in the hypervisor, which after validation can forward them to the unprivileged exception handlers of the

Linux kernel. The hypervisor supplies the kernel exception handlers with some privileged data needed to correctly service an on-going exception (e.g. for pre-fetch abort, the privileged fault address and fault status registers are forwarded to the guest). The exception handlers in the Linux kernel have thus been slightly modified to support this. Among the exceptions that are forwarded to the Linux kernel there are the hardware interrupts delivered by the timer. This allows Linux to implement an internal time based scheduler.

Memory Management To paravirtualize the kernel, we modified the architecture dependent layer of its memory management. In the modified Linux all accesses to the coprocessor registers or to the active page tables are done by issuing the proper hypercalls. The architecture independent layer of the memory management has been left unmodified. In order to speed up the execution of Linux, a minimal emulation layer has been moved from the Linux kernel into the hypervisor itself. This layer reduces the overhead by translating a guest request into a sequence of invocations of the APIs that virtualize the MMU. Since the emulation layer accesses page tables only through the virtualization API, showing memory safety of this component is sufficient to extend the coverage of the verification.

10.2 Run-time Overhead

The port of the Linux kernel 2.6.34 on the hypervisor allows us to present a rough comparison of our approach with existing paravirtualized hypervisors for the ARM architecture [30]. The purpose of the evaluation is more to demonstrate that our approach actually runs with reasonable efficiency. A serious evaluation is out of scope of this work. It requires a more optimised implementation, and a more comprehensive evaluation.

The run-time evaluation is done using LMBench [37] running on Linux 2.6.34 with and without virtualization. The outcome, measured on an ARMv7-A Cortex-A8 system (BeagleBoard-xM [46]), is presented in Table 2. The significant virtualization overhead for the fork benchmarks is due to a large number of simple operations (in this case, write access to a page-table) being replaced with a large number of expensive hypercalls. It may be possible to reduce this overhead with minimal optimisation (e.g. batching). In Table 2 we also report measures from [30], where the authors compare several existing hypervisors for ARM. We point out that these performance numbers have been obtained from different sources, testing different ARM cores, boards and hosted Linux kernels. Hence we do not claim to be able to draw any hard conclusions from these figures about the relative performance of the hypervisors or their underlying architectures. With the purpose of demonstrating that the hypervisor can run efficiently real applications, we also measured the overhead introduced when executing tar, dd and several compression tools.

The second column reports the latency for the version of the hypervisor that aggressively flushes the caches (i.e. the caches are completely clean and invalidated whenever an exception or an interrupt is raised, while the hypervisor in the first column limits cache flushes to the cases of context switch). This naive approach guarantees

Benchmark	Hypervisor	Aggressive cache flushes	L4Linux	Xen	OKL4
null syscall	329%	332%	3043%	150%	60%
read	160%	181%	844%	90%	15%
write	193%	201%	877%	85%	24%
stat	83%	84%	553%		7%
fstat	118%	122%	945%		42%
open/close	121%	119%	433%		
select(10)	78%	84%	4461%		14%
sig handler install	237%	245%	1241%		16%
sig handler overhead	226%	237%	1281%	82%	-14%
protection fault	40%	39%	975%		67%
pipe	168%	3073%	450%	74%	31%
fork+exit	195%	1861%	950%	247%	8%
fork+execve	187%	1787%	591%	239%	5%
pagefaults	435%	8740%	567%		

Table 2: Latency benchmarks. LMBench micro benchmarks for the Linux kernel v2.6.34 running naively on BeagleBoard-xM, paravirtualized on the hypervisor without cache flushing (*Hypervisor*), with aggressive flushing (*Aggressive cache flushes*), and the other hypervisors (*L4Linux*, *Xen*, *OKL4*). Figures in the three last columns have been obtained from different ARM cores, boards and hosted Linux kernels

Applications	Hypervisor	Aggressive cache flushes
tar (500 KB)	0%	171%
tar (1 MB)	0%	108%
dd (10 MB)	100%	1000%
dd (20 MB)	79%	932%
dd (40 MB)	76%	1061%
jpg2gif(5 KB)	0%	117%
jpg2bmp(5 KB)	0%	175%
jpg2bmp(250 KB)	0%	27%
jpg2bmp(750 KB)	-1%	24%
Jpegtrans(270°, 5 KB)	0%	700%
Jpegtrans(270°, 250 KB)	14%	300%
Jpegtrans(270°, 750 KB)	11%	176%
Bmp2tiff(90 KB)	0%	500%
Bmp2tiff(800 KB)	0%	300%
Ppm2tiff(100 KB)	0%	600%
Ppm2tiff(250 KB)	0%	700%
Ppm2tiff(1.3 MB)	50%	350%
Tif2rgb(200 KB)	200%	1100%
Tif2rgb(800 KB)	25%	575%
Tif2rgb(1.200 MB)	31%	462%
sox(atf2wav -r 8000 -bits 16 100 KB)	50%	600%
sox(atf2wav -r 8000 -bits 16 500 KB)	75%	350%
sox(atf2wav -r 8000 -bits 16 800 KB)	83%	267%

Table 3: Latency benchmarks. Application benchmarks for the Linux kernel v2.6.34 running natively on BeagleBoard-xM, paravirtualized on the hypervisor without cache flushing (*Hypervisor*), with aggressive flushing (*Aggressive cache flushes*).

that the actual CPU respects the fully sequential memory model, but introduces severe performance penalties especially in the application benchmarks. Less conservative approaches (e.g. evicting only the necessary physical addresses or forcing the page tables to be allocated in memory regions that are always cacheable) can be adopted for some processor implementations, but they require a more fine-grained modelling including caches and an adaptation of the verification approach for their justification, as discussed in [18].

10.3 Memory Footprint

The main difference between our proposal and the existing verified hypervisors is the MMU virtualization mechanism. The direct paging approach requires a table which contains at most $mem_{size}/block_{size}$ entries, where mem_{size} is the total available phys-

Processes	Direct Paging	Direct Paging	Shadow page table
	256 MB	1 GB	
32	56 KB	224 KB	608 KB
64	64 KB	256 KB	1216 KB
128	72 KB	288 KB	2432 KB

Table 4: Memory footprint. Comparison of memory usage of *Shadow page table* and *direct paging*.

ical memory and $block_{size}$ is the minimum page size (here, 4 KB). Each entry in this table uses $2 + \log_2 max_{ref}$ bits, with the first two bits used to record entry type and max_{ref} being the maximum number of references pointing to the same page. Assuming this number is bound by the number of processes, Table 4 indicates the memory overhead introduced by direct paging.

It should be noted that on ARMv7, most operating systems including Linux dedicate one L1 page to each process and at least three L2 pages to map the stack, the executable code and the heap. Then the OS itself has a minimum footprint of 16 KB + 3*1 KB per process. This footprint is doubled if the underlying hypervisor uses shadow page tables.

11 Evaluation

The hypervisor is implemented in C (and some assembly) and consists of 4529 lines of code (LOC). Excluding platform dependent parts, the hypervisor core is no larger than 2066 LOC. The virtualization of the memory subsystem consists of 1200 LOC. To paravirtualize Linux we changed 1025 LOC of its kernel, 950 in the ARM specific architecture folder and 75 in init/main.c. The paravirtualization is binary compatible with existing userland applications, thus we do not need to recompile either hosted applications or the libc. For comparison, the only other hypervisor that implements direct paging is the Xen hypervisor, which consists of 100 KLOC and its design is not suitable for verification. Instead, the small code base of our hypervisor makes it easier to experiment with different virtualization paradigms and enables formal verification of its correctness. The formal specification consists of 1500 LOC of HOL4 and intentionally avoids any high level construct, in order to make the HOL4 model as similar as possible to the C implementation, at the price of increasing the verification cost. The complete proof consists of 18700 LOC of HOL4.

The verification highlighted a number of bugs in the initial design of the APIs: (i) arithmetic overflow when updating the reference counter, caused by not preventing the guest to create an unbounded number of references to a physical block, (ii) bit field and offset mismatch, (iii) missing check that a newly allocated page table prevents the guest to overwrite the page table itself, (iv) usage of the signed shift operator where the unsigned one was necessary and (v) approval of guest requests that cause unpredictable MMU behaviour. Moreover, the verification of the implementation model identified three additional bugs exploitable by the guest by requesting the validation of page tables outside the guest memory. Finally, the methodology described in Section 9 has been experimented in the verification of the binary code of one of the hypercalls. This experiment identified a buffer overflow in the binary code that was missing in

implementation model. The HOL4 model uses a 10-bit variable to store an untrusted parameter which is later used to index the entries of a page table. The binary code uses a 32-bit registers to store the same parameter, thus causing an overflow when accessing the L2 page table if the received parameter is bigger than 1023. The bug has been fixed by sanitising the input using the mask `parameter = parameter & 0x3ff`.

The project was conducted in three steps. The design, modelling and verification of the APIs for MMU virtualization required nine person months. Here, the most expensive tasks have been the verification of Theorems 1 and 6. The C implementation of the APIs and the Linux port has been accomplished in three person months. While the implementation team was completing the Linux port the verification team started the verification of the refinement, which has taken three months so far. This work is continuing, in order to complete the verification from the HOL4 implementation level down to assembly.

12 Applications

Applications of the hypervisors include the deployment of trusted cryptographic services and trusted controllers. In the first scenario, the hypervisor core is extended with the handlers required to implement message passing. These handlers allow (i) Linux to send a message to the trusted service, (ii) the trusted service to reply with an encrypted message and (iii) the two partitions to cooperatively schedule themselves. The isolation properties guarantee that the untrusted guest cannot access the cryptographic keys stored in the memory of the trusted services. The second scenario includes a device (e.g. a physical sensor) whose IO is memory mapped. The guest is forbidden to access the memory where the IO registers are mapped, thus guaranteeing that the trusted controller is the only subject capable of directly affecting the device. The complete Linux system can be used to provide a rich and complex user interface (either graphical or web based) for the controller logic without affecting its security.

The MMU virtualization solution demonstrated here can be used by other ARM-based software platforms than the hypervisor reported above. A fully fledged hypervisor (e.g. XEN) can use our approach to support hardware that lacks virtualization extensions (e.g. Cortex-A8, Cortex-A5, ARM11). The mechanism can also be used by compiler-based virtual machines and unikernels, which need to monitor the memory configuration and protect it from the rest of the system (e.g. SVA uses a non-verified implementation of direct paging). Customers of cloud infrastructures can also benefit from our approach (see Figure 11a). In this setting, if the virtualization extensions are available, the most privileged execution mode is controlled by the software platform managed by the cloud provider (e.g. a hypervisor). Thus, these extensions cannot be used by the customer to isolate its untrusted Linux from its own trusted services. In this setup, our mechanism can be used to fulfil this requirement.

An interesting application of isolating platforms is the external protection of an untrusted commodity OS from internal threats, as demonstrated in [14]. Trustworthy components are deployed together and properly isolated from the application OS (see Figure 11b). These components are used as an aid for the application OS to restrict its own attack surface, by guaranteeing the impossibility of certain malicious behaviours.

Figure 11: Applications of the secure virtualization platform.

In [11], we show that this approach can be used to implement an embedded device that hosts a Linux system provably free of binary code injection. Our goal is to formally prove that the target system prevents all forms of binary code injection even if the adversary has full control of the hosted Linux and no analysis of Linux is performed.

The system is implemented as a run-time monitor. The monitor forces an untrusted Linux system to obey the executable space protection policy (usually represented as $W \oplus X$); a memory area can be either executable or writable, but cannot be both. The protection of executable space allows the monitor to intercept all changes to the executable code performed by a user application or by the Linux kernel itself. On top of this infrastructure, we use standard signature checking to prevent code injection. Here, integrity of an executable physical block stands for the block having a valid signature. Similarly, the integrity of the system code depends on the integrity of all executable physical blocks. The valid signatures are assumed to be known by the run-time monitor. We refer to this information as the “golden image” (GI) and it is held by the monitor.

We configured the hypervisor to support the following interaction protocol:

1. For each hypercall invoked by a guest, the hypervisor forwards the request to the monitor.
2. The monitor validates the request based on its validation mechanism.
3. The monitor reports to the hypervisor the result of the hypercall validation.

Since the hypervisor supervises the changes of the page tables, the monitor is able to intercept all the memory layout modifications. This makes the monitor able to know whether a physical block is writable: if there exists at least one virtual mapping pointing to such block and having writable access permission. Similarly it is possible to know which physical block is executable.

Then the signature checking is implemented in the obvious way: whenever Linux requests to change a page table (i.e. causing to change the domain of the executable code) the monitor (i) identifies the physical blocks that can be made executable by the request, (ii) computes the block signature and (iii) compares the result with the content

of the golden image. This policy is sufficient to prevent code injection that is caused by changes of the memory layout setting, due to the hypervisor forwarding to the monitor all requests to change the page tables.

Figure 11b depicts the architecture of the system; both the run-time monitor and the untrusted Linux are deployed as two guests of the hypervisor. Using a dedicated guest on top of the hypervisor permits to decouple the enforcement of the security policies from the other hypervisor functionalities, thus keeping the trusted computing base minimal. Moreover, having the security policy wrapped inside a guest supports both the tamper-resistance and the trustworthiness of the monitor. In fact, the monitor can take advantage from the isolation properties provided by the hypervisor. This avoids malicious interferences coming from the other guests (for example from a process of an OS running on a different partition of the same machine). Finally, decoupling the run-time security policy from the other functionalities of the hypervisor makes the formal specification and verification of the monitor more affordable.

The formal model of the system (i.e. consisting of the hypervisor, the monitor and the untrusted Linux) is built on top of the models presented in Section 6.1. Here we leave unspecified the algorithm used to sign and check signatures, so that our results can be used for different intrusion detection mechanisms. The golden image GI is a finite set of signatures $\{s_1, \dots, s_n\}$, where the signatures are selected from a domain S . We assume the existence of a function $sig : 2^{4096*8} \rightarrow S$ that computes the signature of the content of a block. The system behaviour is modelled by the following rules:

$$\frac{\langle \sigma, h \rangle \rightarrow_0 \langle \sigma', h' \rangle}{\langle \sigma, h, GI \rangle \rightarrow_0 \langle \sigma', h', GI \rangle} \quad \frac{\langle \sigma, h \rangle \rightarrow_1 \langle \sigma', h' \rangle \quad \text{validate}(\text{req}(\langle \sigma, h \rangle), \langle \sigma, h, GI \rangle)}{\langle \sigma, h, GI \rangle \rightarrow_1 \langle \sigma', h', GI \rangle} \quad \frac{\langle \sigma, h \rangle \rightarrow_1 \langle \sigma', h' \rangle \quad \neg \text{validate}(\text{req}(\langle \sigma, h \rangle), \langle \sigma, h, GI \rangle)}{\langle \sigma, h, GI \rangle \rightarrow_1 \epsilon(\langle \sigma, h, GI \rangle)}$$

User mode transitions (e.g. Linux activities) require neither the hypervisor nor the monitor intermediation. Theorem 4 justifies the fact that, by construction, the transitions executed by the untrusted component cannot affect the monitor state; (i) the golden image is constant and (ii) the monitor code can be statically identified and abstractly modelled. The executions in privileged mode require the intermediation of the monitor. If the monitor validates the request, then the standard behaviour of the hypervisor is executed. Otherwise the hypervisor performs a special operation to reject the request, by reaching the state that is returned by a function ϵ . Hereafter, the function ϵ is assumed to be the identity. Alternatively, ϵ can transform the state so that the requestor is informed about the rejected operation, by updating the user registers according to the desired calling convention. The function $\text{validate}(\text{req}(\langle \sigma, h \rangle), \langle \sigma, h, GI \rangle)$ represents the validation mechanism of the monitor, which checks at run-time possible violations of the security policies.

To formalize the top level goal of our verification we introduce some auxiliary notations. The “working set” identifies the physical blocks that host executable binaries and their corresponding content. Let σ be a machine state. The working set of σ is defined as

$$WS(\sigma) = \{ \langle bl, \text{content}(bl, \sigma) \rangle \mid \exists pa. mmu_{ph}(\sigma, PL0, pa, ex) \wedge pa \in bl \}$$

By using a code signing approach, we say that the integrity of a physical block is satisfied if the signature of the block’s content belongs to the golden image. Let

$cnt \in 2^{4096 \times 8}$ be the 4 KB content of a physical block bl and GI be the golden image

$$int(GI, bl, cnt) = sig(bl, cnt) \in GI$$

Notice that our security property can be refined to fit different anti-intrusion mechanisms. For example, $int(GI, bl, cnt)$ can be instantiated with the execution of an anti-virus scanner.

The system state is free of malicious code injection if the signature check is satisfied for the whole executable code. That is: Let σ be a machine state, bl be a physical block and GI be the golden image

$$int(GI, \sigma) \Leftrightarrow \forall \langle bl, cnt \rangle \in WS(\sigma) . int(GI, bl, cnt)$$

Finally, in [11] we demonstrate our top level goal: No code injection can succeed.

Theorem 8 *If $\langle \sigma, h, GI \rangle$ is a state reachable from the initial state of the system $\langle \sigma_0, h_0, GI \rangle$ then $int(GI, \sigma)$*

We implemented a prototype of the system. The monitor code consists of 720 lines of C and 100 lines have been added to the hypervisor to support the needed interactions among the hosted components.

13 Concluding remarks

We have presented a memory virtualization platform for ARM based on direct paging, an approach inspired by the paravirtualization mechanism of Xen [7], and the Secure Virtual Architecture [17]. The platform has been verified down to a detailed model of a commodity CPU architecture (ARMv7-A), and we have shown a hypervisor based on the platform capable of hosting a Linux system while provably isolating it from other services. The hypervisor has been implemented on real hardware and shown to provide promising performance, although the benchmarks presented here are admittedly preliminary. The verification is done with respect to a top-level model that augments a real machine state with additional model components. The verification shows complete mediation, memory isolation, and information flow correctness with respect to the top-level model. As the main application we demonstrated how the virtualization mechanism can be used to support a provably secure run-time monitor for Linux that provides secure updates along with the $W \oplus X$ policy.

The main precursor work on formally verified MMU virtualization uses the simulation-based approach of Paul et al [5, 41, 4]. In [5, 41] shadow page tables are used to provide full virtualization, including virtual memory, for “baby VAMP”, a simplified MIPS, using VCC. Full virtualization is generally more complex than the paravirtualization approach studied in the present paper, but the machine model is simplified, information flow security is not supported by the simulation framework, and neither applications nor implementation on real hardware are reported. In [4] the same simulation-based approach is used to study TLB virtualization on an abstract version of the x64 virtual memory architecture. Other related work on verification of microkernels and hypervisors such as seL4 [33] or the Nova project [45] does not address MMU virtualization

in detail. It may be argued that the emergence of hardware based virtualization support makes software MMU virtualization obsolete. We argue that this is not the case. First, many platforms remain or are currently in development that do not yet support virtualization extensions, second, many application hardening frameworks such as Criswell et al. [16], KCoFi [14], Overshadow [10], Inktag [28] and Virtual Ghost [15] rely on some form of MMU virtualization for their internal security, and third, some use cases, e.g. in cloud scenarios, could make good use of software based MMU virtualization to harden VMs without relying on cloud provider hardware.

Our results are not yet complete. The MMU virtualization approach does not support DMA. To securely enable DMA the behaviour of the specific DMA controller must be formally modelled (in [43] the authors describe a framework for such extensions and establish Properties 1 and 2 for the resulting model) and the hypervisor must (i) mediate all accesses to the memory area where the controller’s registers are mapped, (ii) enable a DMA channel only if the pointed physical blocks is *data* and (iii) update the reference counters accordingly. Several embedded platforms are equipped with IOMMUs, that provide HW support to isolate/confine external peripherals that use DMA. However SW based isolation of DMA is still interesting since it can be used in the scenarios where these HW extensions are not available (e.g. CortexM microcontrollers), they are not accessible (e.g. when they are managed by a cloud provider), or in time critical applications since the page walks introduced by the IOMMU can slow down the peripheral and make worst case execution time analysis more difficult.

A tricky problem concern the treatment of unpredictable behaviour in the ARMv7 architecture. The Cambridge ISA model [22] maps transitions resulting in unpredictable behaviour to $\perp$. We ignore this for the following reason. Our verification shows that unpredictable behaviour never arises during hypervisor code execution. This is so since the ARMv7 step theorems used by the lifter are defined only for predictable instructions, and since our invariant guarantees that the MMU configuration is always well defined. As a result unpredictable behaviour can arise only during non-privileged execution, the analysis of which we have in effect deferred to other work [43].

Finally more work is needed to properly reflect caches, TLBs, and, further down the line, multi-core. The soundness of the current implementation depends on the type of data cache, and on flushing the cache when needed, in order to support a linearizable memory model. To enable more aggressive optimisation, and to fully formally secure our virtualization framework on processors with weaker cache guarantees, the model must be extended to reflect cache behaviour.

Acknowledgements.

We thank the anonymous reviewers for their extensive comments. Work partially supported by framework grant “IT 2010” from the Swedish Foundation for Strategic Research, and the CERCES grant from the Swedish Civil Contingencies Agency.

References

- [1] *ARMv7-AR Architecture Reference Manual*. Technical documentation ARM DDI 0406B. ARM Limited, 2008.
- [2] *ARM Security Technology - Building a Secure System using TrustZone Technology*. Technical documentation ARM PRD29-GENC-009492C. ARM Limited, 2009.
- [3] *ARM Cortex-A15 MPCore Processor - Technical Reference Manual*. Technical document ARM DDI 0438I. ARM Limited, 2011.
- [4] E. Alkassar, E. Cohen, M. Kovalev, and W. J. Paul. Verification of TLB virtualization implemented in C. In *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments, VSTTE'12*, pages 209–224, Berlin, Heidelberg, 2012. Springer-Verlag.
- [5] E. Alkassar, M. A. Hillebrand, W. Paul, and E. Petrova. Automated verification of a small hypervisor. In *Verified Software: Theories, Tools, Experiments*, pages 40–54. Springer, 2010.
- [6] A. Azevedo de Amorim, N. Collins, A. DeHon, D. Demange, C. Hrițcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach. A verified information-flow architecture. In *Proceedings of the 41st annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 165–178. ACM, 2014.
- [7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.
- [8] C. Baumann, T. Bormer, H. Blasum, and S. Tverdyshev. Proving memory separation in a microkernel by code level verification. In *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2011 14th IEEE International Symposium on*, pages 25–32. IEEE, 2011.
- [9] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A binary analysis platform. In *Proc. CAV'11*, volume 6806 of *Lecture Notes in Computer Science*, pages 463–469. Springer, 2011.
- [10] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIII*, pages 2–13, New York, NY, USA, 2008. ACM.
- [11] H. Chfouka, H. Nemati, R. Guanciale, M. Dam, and P. Ekdahl. Trustworthy prevention of code injection in Linux on embedded devices. In *Proc. ESORICS*, *Lecture Notes in Computer Science*, page To appear. Springer, 2015.

- [12] D. Cock, Q. Ge, T. Murray, and G. Heiser. The last mile: An empirical study of some timing channels on sel4. 2014.
- [13] E. Cohen, W. Paul, and S. Schmaltz. Theory of multi core hypervisor verification. In *SOFSEM 2013: Theory and Practice of Computer Science*, pages 1–27. Springer, 2013.
- [14] J. Criswell, N. Dautenhahn, and V. Adve. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *IEEE Symposium on Security and Privacy, Oakland*, volume 14, 2014.
- [15] J. Criswell, N. Dautenhahn, and V. Adve. Virtual ghost: Protecting applications from hostile operating systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 81–96, New York, NY, USA, 2014. ACM.
- [16] J. Criswell, N. Geoffray, and V. Adve. Memory safety for low-level software/hardware interactions. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 83–100, Berkeley, CA, USA, 2009. USENIX Association.
- [17] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 351–366, New York, NY, USA, 2007. ACM.
- [18] M. Dam, R. Guanciale, C. Baumann, and H. Nemati. Cache storage channels: Alias-driven attacks and verified countermeasures. In *Security and Privacy (SP), 2016 IEEE Symposium on*, page to appear. IEEE, 2016.
- [19] M. Dam, R. Guanciale, N. Khakpour, H. Nemati, and O. Schwarz. Formal verification of information flow security for a simple ARM-based separation kernel. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security*, pages 223–234. ACM, 2013.
- [20] M. Dam, R. Guanciale, and H. Nemati. Machine code verification of a tiny ARM hypervisor. In *Proceedings of the 3rd International Workshop on Trustworthy Embedded Devices*, TrustedED '13, pages 3–12, New York, NY, USA, 2013. ACM.
- [21] P. B. Daniel and C. Marco. *Understanding the Linux kernel*. " O'Reilly Media, Inc.", 2005.
- [22] A. C. J. Fox and M. O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *proc. ITP'10*, volume 6172 of *Lecture Notes in Computer Science*, pages 243–258. Springer, 2010.
- [23] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Proc. CAV'07*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531. Springer, 2007.

- [24] R. Gu, J. Koenig, T. Ramanandro, Z. Shao, X. N. Wu, S.-C. Weng, H. Zhang, and Y. Guo. Deep specifications and certified abstraction layers. In *ACM SIGPLAN Notices*, volume 50, pages 595–608. ACM, 2015.
- [25] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: End-to-end security via automated full-system verification. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 165–181, Broomfield, CO, Oct. 2014. USENIX Association.
- [26] G. Heiser and B. Leslie. The OKL4 microvisor: Convergence point of microkernels and hypervisors. In *Proceedings of the first ACM Asia-Pacific workshop on Workshop on Systems*, pages 19–24. ACM, 2010.
- [27] C. Heitmeyer, M. Archer, E. Leonard, and J. McLean. Applying formal methods to a certifiably secure software system. *IEEE Trans. Softw. Eng.*, 34(1):82–98, Jan. 2008.
- [28] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. Inktag: Secure applications on an untrusted operating system. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13*, pages 265–278, New York, NY, USA, 2013. ACM.
- [29] G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, 2007.
- [30] A. Iqbal, N. Sadeque, and R. I. Mutia. An overview of microkernel, hypervisor and microvisor virtualization approaches for embedded systems. *Report, Department of Electrical and Information Technology, Lund University, Sweden*, 2110, 2009.
- [31] P. A. Karger and R. R. Schell. Thirty years later: Lessons from the Multics security evaluation. In *ACSAC*, pages 119–126. IEEE Computer Society, 2002.
- [32] N. Khakpour, O. Schwarz, and M. Dam. Machine assisted proof of ARMv7 instruction level isolation properties. In *Certified Programs and Proofs*, pages 276–291. Springer, 2013.
- [33] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *Proc. SOSP'09*, pages 207–220. ACM, 2009.
- [34] D. Leinenbach and T. Santen. Verifying the Microsoft Hyper-V hypervisor with VCC. In *Proc. FM'09*, volume 5850 of *Lecture Notes in Computer Science*, pages 806–809. Springer Berlin Heidelberg, 2009.
- [35] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

- [36] M. K. McKusick and G. V. Neville-Neil. *The design and implementation of the FreeBSD operating system*. Addison-Wesley Professional, 2004.
- [37] L. McVoy and C. Staelin. Lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference, ATEC '96*, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association.
- [38] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. sel4: from general purpose to a proof of information flow enforcement. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 415–429. IEEE, 2013.
- [39] M. O. Myreen, M. J. C. Gordon, and K. Slind. Machine-code verification for multiple architectures - an application of decompilation into logic. In *Formal Methods in Computer-Aided Design, FMCAD 2008, Portland, Oregon, USA, 17-20 November 2008*, pages 1–8, 2008.
- [40] H. Nemati, R. Guanciale, and M. Dam. Trustworthy virtualization of the ARMv7 memory subsystem. In *SOFSEM 2015: Theory and Practice of Computer Science - 41st International Conference on Current Trends in Theory and Practice of Computer Science, Pec pod Sněžkou, Czech Republic, January 24-29, 2015. Proceedings*, pages 578–589, 2015.
- [41] W. Paul, S. Schmaltz, and A. Shadrin. Completing the automated verification of a small hypervisor-assembler code verification. In *Software Engineering and Formal Methods*, pages 188–202. Springer, 2012.
- [42] R. Richards. Modeling and security analysis of a commercial real-time operating system kernel. In D. S. Hardin, editor, *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 301–322. Springer US, 2010.
- [43] O. Schwarz and M. Dam. Formal verification of secure user mode device execution with DMA. In *Hardware and Software: Verification and Testing*, pages 236–251. Springer, 2014.
- [44] T. A. L. Sewell, M. O. Myreen, and G. Klein. Translation validation for a verified OS kernel. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 471–482. ACM, 2013.
- [45] U. Steinberg and B. Kauer. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *Proceedings of EuroSys*, 2010.
- [46] The BeagleBoard.org Foundation. The BeagleBoard-xM, 2010.
- [47] J. Tinnes. Linux null pointer dereference due to incorrect proto-ops initializations (cve-2009-2692), 2009.
- [48] P. Varanasi and G. Heiser. Hardware-supported virtualization on ARM. In *Proceedings of the Second Asia-Pacific Workshop on Systems, APSys '11*, pages 11:1–11:5, New York, NY, USA, 2011. ACM.

- [49] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta. Design, implementation and verification of an extensible and modular hypervisor framework. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy, SP '13*, pages 430–444, Washington, DC, USA, 2013. IEEE Computer Society.
- [50] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *ACM SIGOPS Operating Systems Review*, volume 27, pages 203–216. ACM, 1994.
- [51] M. M. Wilding, D. A. Greve, R. J. Richards, and D. S. Hardin. Formal verification of partition management for the AAMP7G microprocessor. In *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 175–191. Springer US, 2010.
- [52] L. Zhao, G. Li, B. De Sutter, and J. Regehr. ARMor: Fully verified software fault isolation. In *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*, pages 289–298. IEEE, 2011.