Formal Verification of Secure User Mode Device Execution with DMA

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

[1] SICS Swedish ICT, Kista, Sweden
[2] KTH Royal Institute of Technology, Stockholm, Sweden
{oschwarz,mfd}@kth.se

Legal Notice. This is the author version of the correspondent paper published in "Hardware and Software: Verification and Testing", the proceedings of HVC 2014 (editor: Eran Yahav), Springer LNCS 8855. The publisher is Springer International Publishing Switzerland. The final publication is available at
`http://link.springer.com/10.1007/978-3-319-13338-6_18`.

Abstract. Separation between processes on top of an operating system or between guests in a virtualized environment is essential for establishing security on modern platforms. A key requirement of the underlying hardware is the ability to support multiple partitions executing on the shared hardware without undue interference. For modern processor architectures - with hardware support for memory management, several modes of operation and I/O interfaces - this is a delicate issue requiring deep analysis at both instruction set and processor implementation level. In a first attempt to rigorously answer this type of questions we introduced in previous work an information flow analysis of user program execution on an ARMv7 platform with hardware supported memory protection, but without I/O. The analysis was performed as a semi-automatic proof search procedure on top of an ARMv7 ISA model implemented in the Cambridge HOL4 theorem prover by Fox et al. The restricted platform functionality, however, makes the analysis of limited practical value. In this paper we add support for devices, including DMA, to the analysis. To this end, we propose an approach to device modeling based on the idea of executing devices nondeterministically in parallel with the (single-core) deterministic processor, covering a fine granularity of interactions between the model components. Based on this model and taking the ARMv7 ISA as an example, we provide HOL4 proofs of several noninterference-oriented isolation properties for a partition executing in the presence of devices which potentially use DMA or interrupts.

Keywords: peripheral devices, DMA, separation, isolation, user mode execution, ARM, formal hardware/software co-verification, theorem proving, HOL4

1 Introduction

Modern computing platforms usually execute multiple kinds of services together. Entertainment software runs next to online-banking applications. Personal communication services run next to business software. For security, there is a strong need to execute processes in isolation from each other, such that mutual influence is minimized and their integrity and confidentiality fully protected. Some approaches attempt to achieve this level of isolation within the commodity operating system, while others base upon separation kernels, micro kernels or virtualization. In all cases, the hardware platform is required to allow for strong compartmentalization of process execution. Untrusted processes should neither be able to influence processes at higher trust levels nor to learn anything about their state of execution. Basic protection is enabled by several privilege rings of operation and memory protection/management units (MPU/MMU), controlled by control registers, coprocessors and configurations in memory. Information can potentially flow via multiple system components and operations, such as memory accesses by the CPU, directly accessible registers, side effects of control registers, coprocessors, timing channels, device ports, device accesses to memory, or interrupts, to just name a few. Therefore it is crucial to understand and document the information flows that are possible on a complex platform. These flows are not always obvious. For example, on some x86 processors it is possible for low-privilege code to overwrite higher privilege code by writing to an address that usually refers to the video card [5]. To enable this attack, it suffices to flip a configuration bit usually accessible from the low privilege level. On ARM, comparison instructions change flags in the current program status register (CPSR). When switching processes, those flags therefore need to be cleared or reloaded from the register bank of the invoked process. Peripheral devices further increase a system's complexity. Assigning them to only one process per device is sometimes insufficient to prevent information flow between processes. If a device has the capability of performing direct memory access (DMA), it can be programmed to circumvent the access policy of the MMU unless advanced hardware support for virtualization is provided and this support is soundly configured, which is by no means self-evident. Even if the configuration of DMA controllers is monitored to prevent copying between partitions, undesired information flows can still occur. For example, a device can fire an interrupt depending on the content of memory controlled by a user process, allowing for side channel communication based on the delays introduced by such interrupts. Given the complexity of modern hardware, it is not trivial to avoid misconfiguration. In previous work [10] we introduced a formal information flow analysis of ARMv7 user mode execution on instruction set architecture (ISA) level, however, not yet covering devices. With devices, the system's state increases and so does the set of possible information flows. CPU and multiple DMA devices with unknown behaviour can execute in parallel, possibly accessing the same memory, with an unknown interleaving.

This paper presents the following contributions. First, we extend the Cambridge HOL4 model of the ARM architecture [7] by a general device framework. To the best of our knowledge, this is the first theorem prover model for de-

vices capable of reasoning on DMA. It is sufficiently detailed to capture possible information flows on modern systems. The adaptation to other processor architectures can be done with a minor effort. Second, we identify several secure device configurations. Since the focus is on platform information flow security rather than functionality, we do not restrict verification to concrete device specifications, but provide a suitable abstraction. For the verification of a system's separation properties, it is then sufficient to show that the configuration of the system devices complies with the identified abstract requirements. Finally, based on the proposed configurations and the device framework, we prove the following partitioning-related properties of the ARMv7 architecture with devices:

1. *Non-infiltration* states that the user mode execution of an ARMv7 processor is independent of devices that neither write to the memory accessed by the active process nor fire interrupts.
2. The integrity property of *extended non-exfiltration* states in turn that user mode processes are unable to influence devices that do not read from CPU-modifiable memory. Moreover, other protected resources[3], such as memory of neighboring processes, can not be modified by the process. That is true even if dedicated peripheral devices do access these resources in parallel. More specifically, the transformation of these resources depends only on such dedicated and inaccessible devices.
3. The third property, *filtered device non-infiltration*, states that devices which operate on disjunct resources can not influence each other without the interaction of the CPU.

One of the added challenges in the formulation and verification of the properties compared to [10] is that - with CPU and devices operating in parallel - different principles can cause different effects on the shared state. Covering separation during user mode execution, the results can be applied in the verification of hypervisors, separation kernels and operating systems. To the best of our knowledge, this is the first work on non-interference like platform properties for autonomous device execution.

2 Related Work

Hillebrand et al. [9] describe a pen and paper model, later formalized in Isabelle/HOL [1], for a memory-mapped hard disk integrated with a RISC architecture. The model includes side effects on device port reads/writes, interrupts and an external environment. The latter is also used to realize non-determinism, especially in timing matters. Direct memory access is not covered. Furthermore, unlike ARM, the processor model does not perform multiple memory operations per instruction (instruction fetches are assumed to not refer to device ports), which allows for executing processor and device steps in an interleaved way after one another, without considering device progress within a single processor step.

[3] See Section 6.3 for the complete list of protected resources.

In [1] they describe the exploitation of an oracle that enables the sequentialization of the concurrent execution of devices and CPU. While the concrete order of events in a system is hard to predict, this oracle allows for the quantification over all execution orders and external inputs. These results were applied in the functional correctness verification of a microkernel [2]. The system architecture includes concurrent devices; besides a hard disk used for page fault handling also devices accessible by user processes are considered. Using refinement techniques, the authors were able to establish a simulation relation between an abstract microkernel programming framework and the instruction level. On the abstract levels devices are represented as ghost data structures.

Duan and Regehr [4] describe a general device model framework integrated with the HOL4 model for ARM6 by Anthony Fox [6] in a lock-step manner. They provide a proof of concept for a UART device and its driver, presenting statements on functionality, (memory) safety and timing. Similar to [9] and [1], they model side effects of memory mapped accesses to device ports and exploit input streams. Again, DMA is not supported. The authors prove that the integration of new devices to the system does not cause new undefined behaviour and preserves already established system predicates. This allows to verify driver correctness for one device at a time, but clearly would not hold for DMA devices. In his PhD thesis [3], Duan integrates his model into the Cambridge model of ARMv7 and adds reasoning about interrupts. Since ARMv7 has instructions that perform multiple memory accesses, device port reads/writes have been integrated into the primitives for memory accesses. Also autonomous device transitions are integrated into the execution cycle, however, they occur only once per instruction. In a DMA setting this is not sufficient since physical memory can be changed by devices between two memory accesses from the CPU side. In order to reason about DMA with a finer granularity and to allow for non-deterministic device progress, we propose a different model in Section 5.

Monniaux modelled a USB controller in C and used an extended version of the Astrée static analyzer to verify that neither controller nor its driver will transfer data incorrectly [11]. He includes asynchronous DMA into his reasoning. By modelling the controller's behaviour with non-deterministic choices, an over-approximation is achieved. Isolation from untrusted software is not discussed.

XMHF [12] is a hypervisor framework for x86 exploiting virtualization support, in particular the DMA protection of Intel Vt-d and AMD's device exclusion vectors. The framework allows unmodified guests direct device access. System devices are included in the attacker model. Exploiting the model checker CBMC, mainly memory integrity is verified. As for direct memory access, CBMC verifies that the control register value written to the DMA protection hardware register has the bit set which enables DMA protection. The DMA table is manually audited. However, it seems that the actual effects of devices or the DMA protection unit are not part of the model. In the present paper we focus on systems without hardware support for virtualization.

$$\begin{aligned}\texttt{arm_state} \ = < | \ & \texttt{psrs} && : \texttt{PSRName} \rightarrow \texttt{ARMpsr}; \\ & \texttt{regs} && : \texttt{RName} \rightarrow \texttt{word32}; \\ & \texttt{memory} && : \texttt{word32} \rightarrow \texttt{word8}; \\ & \texttt{coproc} && : \texttt{coprocessors}; \\ & \texttt{accesses} && : \texttt{memory_accesslist}; \\ & \texttt{misc} && : \texttt{Monitors} \ \# \ \texttt{ARMinfo} \ \# \ \texttt{bool} \ \# \ \texttt{bool} \ | >;\end{aligned}$$

Fig. 1. The ARM state in HOL4

The properties shown in this paper are inspired by Heitmeyer et al. [8], who formulated non-infiltration and non-exfiltration for a separation kernel. We adapt those properties to a platform with DMA devices and a CPU in user mode.

3 The HOL4 ARM Model

We base our work on Fox et al's monadic HOL4 ISA model [7] of ARMv7 platforms without hardware extensions such as TrustZone or virtualization support. Figure 1 shows a simplified definition of the processor state in this model. The function `psrs` returns the value of a processor state register (of type `ARMpsr`). The processor state registers include the current program status register, `CPSR`, in addition to the banked psrs `SPSR_m` for each privileged mode `m`, except for system mode. The ARMv7 core provides seven processor modes: one non-privileged user mode `usr`, and six privileged modes, activated when an exception (such as an interrupt) is invoked. The function `regs` takes a register name and returns its value. The ARM registers include sixteen general purpose registers ($\texttt{r0} - \texttt{r15}$) that are available from all modes in addition to the banked registers of each privileged mode that are available only in that mode. The function `memory` maps an address (`word32`) to a byte (`word8`). Caches are not represented in the model. The field `coproc` represents the set of coprocessor registers in `CP14` and `CP15` implicitly influencing execution, to a large extent even user-mode/exception execution. The field `misc` represents exclusive monitors for synchronization purposes, general information about the state, e.g. the architecture version, if the system is waiting for an interrupt etc, and `accesses` records the accesses to the memory.

A *computation* in the monadic HOL4 ARM model is a term of the type

$$\alpha \ \texttt{M} = \texttt{arm_state} \mapsto (\alpha, \texttt{arm_state}) \ \texttt{error_option}$$

where `error_option` is a datatype defined as:

$$(\alpha, \beta) \ \texttt{error_option} = \texttt{ValueState of} \ \alpha \Rightarrow \beta \ | \ \texttt{Error of string}$$

Computations act on a state (`arm_state`) and return either `ValueState` $a\ s$, a new state s along with a return value a of type α, or an error e (if the computation is underspecified by the ARM specification). The monad unit `constT` injects a value into a computation, i.e. `constT` $a\ s =$ `ValueState` $a\ s$, while binding is a

sequential composition operation

$$f_1 \gg\!=_e f_2 = \lambda s.\texttt{case}\ f_1\ s\ \texttt{of Error}\ c \rightarrow \texttt{Error}\ c$$
$$\mid \texttt{ValueState}\ a\ s' \rightarrow \texttt{if}\ e\ s'\ \texttt{then}\ f_2\ a\ s'\ \texttt{else}\ f_1\ s.$$

That is, if e holds in the final state of f_1, the return value of f_1 is passed to f_2 as the input parameter, otherwise f_2 is not executed. In addition to unit and binding, the ARM monadic specification uses standard constructs for lambda, full conditional, `let`, and `case`, as well as the monad operations parallel composition, positive conditional (`condT` $e\ f$ = `if` e `then` f `else constT` ()), error (`errorT` a = `Error` a), and an iterator. Values of state components can be obtained and set by `readT` $f = \lambda y.$(`ValueState` $(f\ y)\ y$) and `writeT` $f = \lambda y.$(`ValueState` () $(f\ y)$).

4 Memory Management

The Memory Management Unit (MMU) enforces memory access policies and is therefore crucial for isolation. MMU configurations consist of page tables in memory and dedicated registers of `CP15`. Specific to ARM is the possibility of partitioning pages into collections of memory regions (*domains*), each representing one security role. The coprocessor registers involved are `SCTLR`, `TTBR0` and `DACR`. The `SCTLR` register determines whether the MMU is enabled, `TTBR0` contains the base address of the page table, and `DACR` manages the ARM domains.

In [10] we extended the ARM model with MMU functionality. The extended model defines two key functions, `permitted`, to account for access permissions, and `mmu_setup`, to reflect a "good configuration" property. The permission evaluation function `permitted` $a\ b_w\ (v_s,\ v_t,\ v_d)\ b_p\ m$ takes as parameters an address a, a flag b_w indicating whether reading or writing access is to be evaluated, the values of `SCTLR`, `TTBR0` and `DACR`, a flag b_p indicating whether permissions are to be checked against a privileged mode, and the memory m containing the page tables. The pair of booleans returned by `permitted` states whether the access permission on the specified location is defined in the given configuration, and the outcome of that decision (`true` if access is granted). Here, we apply a basic version of `permitted`, supporting one-level page tables with an identity address translation, but including the interpretation of ARM domains. It is shown that `permitted` is defined for all addresses in all reachable states.

The history of memory accesses is tracked in the `accesses` ghost field of the machine state, allowing to compute the set of memory locations accessed by an instruction. To stop computation after the first access violation, $\gg\!=_{\texttt{nav}}$ has been chosen as standard binding operator. The property `nav` s holds if there is no access violation recorded in state s. Formally, this is the case if there is no entry in the access list of machine state s that causes `permitted` to return a negative answer in the current configuration of s. The recording of an access always happens before the access itself.

We finally need to formulate a suitable well-formedness condition for the MMU configuration. Let `accessible` $i\ a$ express that address a is readable and writable by user process i. Other, more refined, static user level access policies

can be supported with minor effort. The predicate $\texttt{mmu_setup}\ i\ s$ holds if (i) the MMU configuration $((d,p) = \texttt{permitted}\ a\ b_w\ (\texttt{mmu_registers}\ s) \perp s.\texttt{memory})$ for any address a and access type b_w is defined (i.e., d is $\texttt{true}$), (ii) the state s implements the desired access policy for process i (i.e., $p = \texttt{accessible}\ i\ a$), and (iii) none of the active page tables in s (represented by the address set $\texttt{page_table_adds}\ s$) is accessible according to the policy.

$$\begin{aligned}&\texttt{mmu_setup}\ i\ s :=\\ &\forall a,w,d,p.\ ((d,p) = \texttt{permitted}\ a\ b_w\ (\texttt{mmu_registers}\ s) \perp s.\texttt{memory})\\ &\quad \Rightarrow d \wedge (p = \texttt{accessible}\ i\ a)\ \wedge\ (a \in (\texttt{page_table_adds}\ s) \Rightarrow \neg\texttt{accessible}\ i\ a)\end{aligned}$$

For the properties shown in Section 6 we furthermore prohibit user space processes to access device ports by assuming that the (state-independent) set of device addresses and `accessible` addresses are disjoint for every user process.

5 Device Model Framework

We present a general device model framework, capable of reasoning on DMA devices and with the ambition to cover all possible executions of a platform where the single-core processor and multiple devices run in parallel. In practice, changes to shared resources such as memory happen asynchronously and in a practically unpredictable order. We apply a non-deterministic approach that takes into account all possible interleavings and - to be conservative on timing behaviour - all possible durations of device and CPU actions, without restrictions on deadlines. Naturally, this does not allow to reason on whether an operation will be finished before a certain event or not. A timing accurate model would need to take CPU and system implementation specific details into account, including caches, MMU implementation specifics (such as the translation lookaside buffer), pipeline architecture and bus contention protocols. Models at this level of detail are surely interesting, but likely to be vastly more complex. The main challenge when integrating DMA into a device model is that memory can potentially change at any time, for example, between reading two words belonging to a multiple load instruction. Also inter-device communication can occur in any order and granularity. This precludes models that synchronize CPU and devices only between different CPU instructions. To address this challenge and allow for asynchronous device execution, we augment the CPU model with an abstract scheduler as suggested in [1], an oracle of the type

$$\texttt{oracle} : \texttt{num} \rightarrow (\texttt{dev_name}\ \#\ \texttt{word32 option})\ \texttt{option}$$

The oracle provides a non-deterministic sequence of activity entries where the n-th activity entry $\texttt{oracle}\ n$ is either `NONE` (then the CPU is progressed rather than a device) or a tuple $\texttt{SOME}\ (d, e_{iopt})$, indicating the device with identifier (`dev_name`) d to progress one step, possibly in the context of the optional external 32-bit input e_{iopt}. We assume processor liveness: $\forall n.\ \exists m.\ (m \geq n) \wedge (\texttt{oracle}\ m = \texttt{NONE})$. Liveness of devices can be optionally included, but is not required for the properties we show in this paper. To include devices into

the machine state, `arm_state` is extended by the following components:

```
devices   : dev_rec;
ext_out   : dev_name → word32 list;
int_fired : bool;
counter   : num
```

The record `devices` subsumes the states of all devices [4]. The external output is represented by a finite stream of 32-bit words for each device, accessible via the map `ext_out`, mapping each device identifier to the list of outputs produced so far for that device. Whether an interrupt has been fired during the current execution cycle is stored in `int_fired`. Fast interrupts or advanced interrupt controllers are not part of the model. Finally, `counter` points to the current position in the oracle index and is incremented every time the oracle is invoked. Devices can progress in one of four ways:

- *Autonomously*: A device may make processor-independent progress, either by entirely internal actions or by receiving external inputs, accessing memory, raising interrupts, or producing external outputs. The function

  ```
  progress : device ↦ (word32 option)
    ↦ (mem_req option # bool # word32 option # device) option
  ```

  takes as arguments a device state D and a possible external input e_{iopt}. It returns either an "error" (`NONE`) representing undefined behaviour or a tuple $(r_{opt}, b_{int}, e_{oopt}, D')$ with an optional read/write access request r_{opt} to the system's memory bus (including an address and the access type), a flag b_{int} indicating a possible interrupt, an optional external output e_{oopt} and the updated device state D'. This function is used to progress devices with a non-deterministic frequency after every executed CPU instruction and between memory accesses made by the CPU or other devices.
- *Upon reception of a pending reply to a memory bus read*: As a result of an autonomous step, a device can send a read request to the bus, in order to read from the system memory or from the port of another device. The result is communicated to the device by invoking the `receive` operation:

  ```
  receive : device ↦ mem_req ↦ mem_answer ↦ device option
  ```

  For a given device state D and request r being answered, `receive` $D\ r\ v$ passes the read value v (as either byte or word) to D and returns either an error (`NONE`) or the updated device D'. Write operations requested by devices do not have an answer and thus change only the memory, but not the device. We assume that reads are atomic operations and the memory bus will always complete an issued read before handling new operations. In other words, we exclude scenarios where a device notices that one of its ports is being read and already starts side effect computations affecting memory or other system components without first returning the requested value.

[4] We notate `devices.`d for the state of the device with identifier d in the record `devices`.

```
advance_single f n := readT (λs. s.devices) ≫=T
                       (λD̂. (case oracle n of NONE ⇒ constT ()
                        |SOME (d, e_iopt) ⇒
                         condT (f d)
                           (case progress D̂.d e_iopt of NONE ⇒ errorT ε
                            |SOME (r_opt, b_int, e_oopt, D′) ⇒
                              update_device d D′ ≫=T
                              (λu. update_output d e_oopt ≫=T
                              (λu. condT b_int
                                (writeT (λs. s with int_fired := T)) ≫=T
                                (λu. case r_opt of NONE ⇒ constT ()
                                  |SOME r ⇒ mem_acc_by_dev r d))))) ≫=T
                       (λu. increment_counter))
```

Fig. 2. The `advance_single` computation.

That is no limitation for the properties we show in this paper, since we do not consider port accesses in them. As for reads from physical memory, for any race condition outcome there is always one initiation of the oracle that represents this outcome within the model.

- *As side effect on port reads*: The CPU or another device may read from an address that is mapped to a device. This address can belong to a device register, but in general it is not required that such a register is physically existing, for example when the address is associated with a side effect. We therefore use the general term *port* rather than register. We assume atomic 32-bit accesses to device ports and that port accesses are not cached. The function

  $$\texttt{d_read} : \texttt{device} \mapsto \texttt{word32} \mapsto (\texttt{word32}\ \#\ \texttt{device})\ \texttt{option}$$

  takes as arguments a device state D and the port number indicating which port of the device is to be read. A special data structure of the model maps any virtual address to either physical memory or a device identifier together with a port number. The result of `d_read` is either `NONE` or the read 32-bit value together with a possibly updated device state D'.
- *As side effect on port writes*: the function

  $$\texttt{d_write} : \texttt{device} \mapsto \texttt{word32} \mapsto \texttt{word32} \mapsto \texttt{device option}$$

  takes as arguments a device state D, the port number indicating which port of the device is to be written to and the 32-bit value to be written. It returns either an error (`NONE`) or the updated device state D'.

Different types of devices will have different behaviour. That is, the concrete functionalities of the device functions depend on the addressed device. While `d_write` and `d_read` are integrated into the existing memory access primitives of the ARM model (similar to [3]), `progress` and `receive` are used to realize

autonomous progress of devices. Figure 2 defines `advance_single` f n that uses the oracle at position n to determine the next device to progress autonomously and that updates the state with the effect of this progress accordingly. Subsequently, resulting memory requests are realized (including possible side effects when directed to other devices) and finally `counter` is increased. A filtering predicate f can be used to apply those steps only to devices d for which f d holds. Here, `update_device` and `update_output` update the `devices` and `ext_out` components of the current state, respectively, and `mem_acc_by_dev` r d realizes the memory access request r on behalf of device d. Our model does not include any IOMMU. The repeated execution of `advance_single` is realized by `advance`, where for $n > 0$, `advance` f n traverses the oracle with filtering predicate f up to oracle position n and `advance` f 0 traverses the oracle until a `NONE` as activity entry indicates that execution will continue on the CPU side. The `advance` computation will synchronize devices and CPU before each memory bus access (for memory mapped ports and physical memory) of the CPU[5] and additionally between two execution cycles. The model supports instruction fetching from device addresses, but we assume that page table walks are not performed on device ports. In the properties shown in this paper we assume an MMU setting that prohibits both, by choosing device addresses, page table addresses and user space accessible memory to be disjoint.

Incorporating the MMU and device extension, the instruction execution function `next` (Fig. 3) involves the following functionality: if an interrupt is pending and not masked, an interrupt exception is taken. Otherwise, the CPU may (if requested so by the previous instruction) wait for an interrupt or fetch and execute the next instruction pointed to by the program counter. If an access violation is recorded during instruction fetching or execution, a prefetch or data abort exception is initiated. The access list is cleared between the single steps and unconditional binding $\gg\!=_{\mathtt{T}}$ is used occasionally, preventing the execution from halting and instead allowing the initiation of exceptions and the detection of possible further violations. In addition to the synchronization phases before any of the CPU's memory operations, possible autonomous device steps are considered after each instruction execution, in order to account for interrupt initiations.

As discussed earlier, our model is not clock accurate. While this is common with related work, usually a fixed duration is assumed for all instructions [3]. In our model, durations are non-deterministic, controlled by the oracle. However, given a specific oracle sequence, memory extensive instructions generally consume more oracle entries (i.e, time). For the properties of this paper and the targeted abstraction level, concrete instruction time is not relevant.

6 Security Properties

We next turn to formalizing several partitioning properties in terms of non-infiltration and non-exfiltration (cf. [8]), adapted to our setting, i.e., arbitrary

---

[5] Consequently, accesses to the shared state, in particular the memory bus, determine the granularity of the system.

```
next := (clear_alist ≫=
        (λu. readT (λs. s.int_fired ∧ ¬s.psrs(0,CPSR).I) ≫=
        (λb. if (¬b) then
                waiting_for_interrupt ≫=
                (λw. condT (¬w)
                        (fetch_instruction ≫=T
                        (λ(o,i). is_viol ≫=T (λa. clear_alist ≫=
                        (λu. if a then prefetch_abort
                             else (execute i ≫=T (λu. is_viol ≫=T
                                  (λa. condT a
                                       (clear_alist ≫=
                                       (λu. data_abort))))))))) ≫=T
                (λu. advance all 0)
            else take_irq_exception ≫= (λu. clear_interrupts))))
```

Fig. 3. The `next` computation.

and unknown user mode code executing on an ARMv7 CPU and in parallel with DMA devices. The isolation does not rely on an IOMMU. Together with a proper separation kernel (configuring devices, mediating user registers etc.) the discussed properties allow for establishing full process isolation within a system.

6.1 Suitable Device Configurations

Since isolation between CPU and DMA devices requires controlled device behaviour, we first describe possible device configurations that we consider secure. They allow the devices to change their internal state in an arbitrary way, but impose restrictions on DMA and interrupts. Kernels are responsible for realizing such a device configuration, in order to guarantee that process isolation is maintained when yielding to user mode. We expect those configurations to stay preserved throughout the whole user mode execution (while access to device ports is forbidden to both CPU and other devices). Formally, a configuration C is called *invariant* if it is preserved over autonomous steps, including the reception of replies to autonomously issued read requests:

$$\begin{aligned}&\texttt{invariant}\ C := \ \forall D.\ C\ D\ \Rightarrow\\ &\quad(\forall e_{iopt},\ r_{opt},\ b_{int},\ e_{oopt},\ D'.\\ &\qquad(\texttt{progress}\ D\ e_{iopt} = \texttt{SOME}\ (r_{opt}, b_{int}, e_{oopt}, D')) \Rightarrow C\ D')\\ &\quad\wedge (\forall r,\ v,\ D'.\ (\texttt{receive}\ D\ r\ v = \texttt{SOME}\ D') \Rightarrow C\ D')\end{aligned}$$

A property P holds on a device D in a *stable* way if it is established by an invariant configuration C:

$$\texttt{stable}\ P\ D := \exists C.\ \texttt{invariant}\ C\ \wedge\ C\ D \wedge\ (\forall D'.\ C\ D' \Rightarrow P\ D')$$

The stable properties we are interested in guarantee that devices are configured in a way that prevents them from communicating with other devices, running into an undefined state, accessing memory out of well-defined boundaries or firing interrupts in dependency on DMA operations. We believe that many devices

(e.g., timers or DMA controllers) can be configured to respect those restrictions. The `restricted_dma` predicate holds if a device is configured to restrict its DMA requests to a set A of memory addresses.

$$\texttt{restricted_dma}\ A\ D := \forall e_{iopt},\ r,\ b_{int},\ e_{oopt},\ D'.$$
$$(\texttt{progress}\ D\ e_{iopt} = \texttt{SOME}\ (\texttt{SOME}\ r, b_{int}, e_{oopt}, D')) \Rightarrow (\texttt{access_request_map}\ r \subseteq A)$$

Here, `access_request_map` maps a memory request to the set of byte addresses it involves. A device is called `silent` if A does not include device ports. Devices not firing interrupts are called `interrupt_free`. We say that a device is `errorfree`, if `progress` and `receive` do not return `NONE` for any inputs. Based on those properties, we distinguish three specific device configurations: devices involving DMA operations on the memory of the active process, devices involving DMA operations on the memory of other processes, and devices that are allowed to fire an interrupt.

$$\begin{aligned}
&\texttt{own_devices}\ i\ D := \texttt{stable}\ (\texttt{restricted_dma}\ (\texttt{own_add}\ i))\ D \\
&\qquad\qquad \wedge\ \texttt{stable}\ \texttt{interrupt_free}\ D \\
&\texttt{foreign_devices}\ i\ D := \texttt{stable}\ (\texttt{restricted_dma}\ (\texttt{foreign_add}\ i))\ D \\
&\qquad\qquad \wedge\ \texttt{stable}\ \texttt{interrupt_free}\ D \\
&\texttt{interrupt_devices}\ D := \texttt{stable}\ (\texttt{restricted_dma}\ \texttt{empty_set})\ D
\end{aligned}$$

Here, `own_add` i is the set of addresses belonging to process or partition i, while `foreign_add` i spans exactly over the other user partitions. We do not allow a device to do both, accessing memory and firing interrupts. This is to prevent information flow from a user process' memory to another process' perception of execution time. [6] For a given user process i we assign each device d to one of three classes, `OWN` i d, `FOREIGN` i d or `INTERRUPT` d, that correspond to the configurations `own_devices` i D, `foreign_devices` i D and `interrupt_devices` D, respectively. While configurations refer to a concrete device state D, device classes are state-independent. We require that each device is in at least one of the three classes. The system properties discussed in the following subsections have a correct configuration of the devices as a prerequisite. The configuration of each device in the current state is supposed to follow the specification of the given class. Moreover, devices are not allowed to communicate with other devices or to run into an underspecified state.

$$\begin{aligned}
&\texttt{device_setup}\ i\ s := \forall d. \\
&\quad (\texttt{OWN}\ i\ d \Rightarrow \texttt{own_devices}\ i\ s.\texttt{devices}.d) \\
&\quad \wedge\ (\texttt{FOREIGN}\ i\ d \Rightarrow \texttt{foreign_devices}\ i\ s.\texttt{devices}.d) \\
&\quad \wedge\ (\texttt{INTERRUPT}\ d \Rightarrow \texttt{interrupt_devices}\ s.\texttt{devices}.d) \\
&\quad \wedge\ \texttt{stable}\ \texttt{errorfree}\ s.\texttt{devices}.d\ \wedge\ \texttt{stable}\ \texttt{silent}\ s.\texttt{devices}.d
\end{aligned}$$

6.2 Non-infiltration

Confidentiality of the kernel and neighboring user processes (including their devices) and the integrity of the active user process is guaranteed by non-

[6] Alternative configurations could allow DMA devices to fire interrupts, as long as those interrupts are masked while foreign processes are executing. However, this requires a very careful and more complex design at kernel level to avoid timing channels when interrupts occur close to context switches.

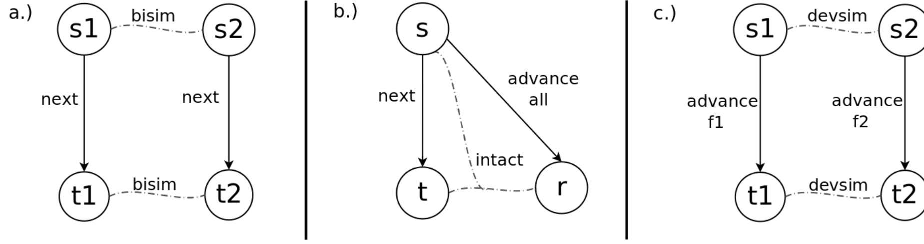

Fig. 4. a.) non-infiltration, b.) extended non-exfiltration, c.) filtered device non-infiltration

infiltration, a noninterference-like property at the user mode single instruction level. Consider two machine states in user mode that are *low equivalent* in the sense that the two states agree on the resources (devices, registers and memory) that are permitted to influence user mode execution, but do not necessarily agree on other resources. Non-infiltration (Fig. 4.a) holds if the poststates, after execution of one instruction, remain low equivalent (or produce the same error).

Theorem 1. *Non-infiltration*

$$\begin{array}{l}\forall s_1,\ s_2,\ i.\ (\texttt{mode}\ s_1 = \texttt{mode}\ s_2 = \texttt{usr})\ \wedge\ \texttt{mmu_setup}\ i\ s_1\ \wedge\ \texttt{mmu_setup}\ i\ s_2\\ \quad\wedge\ \texttt{device_setup}\ i\ s_1\ \wedge\ \texttt{device_setup}\ i\ s_2\ \wedge\ \texttt{bisim}\ i\ s_1\ s_2\\ \Rightarrow (\exists t_1,\ t_2.\ (\texttt{next}\ s_1 = \texttt{ValueState}\ ()\ t_1\ ) \wedge\ (\texttt{next}\ s_2 = \texttt{ValueState}\ ()\ t_2)\\ \quad\wedge\ \texttt{bisim}\ i\ t_1\ t_2)\ \vee\ (\exists e.\ (\texttt{next}\ s_1 = \texttt{Error}\ e)\ \wedge\ (\texttt{next}\ s_2 = \texttt{Error}\ e))\end{array}$$

The relation `bisim` is the low equivalence relation. User mode processes are allowed to be influenced by the user mode registers, the memory assigned to them, devices with access to that memory, interrupt devices, the `CPSR`, the coprocessors, pending access violations and the `misc` state component. Formally:

$$\begin{array}{l}\texttt{bisim}\ i\ s_1\ s_2 :=\\ \quad (s_1.\texttt{counter} = s_2.\texttt{counter})\ \wedge\ (s_1.\texttt{int_fired} = s_2.\texttt{int_fired})\\ \ \wedge\ \texttt{equal_user_regs}\ s_1\ s_2\ \wedge\ (\forall a.\ \texttt{accessible}\ i\ a \Rightarrow (s_1.\texttt{memory}\ a = s_2.\texttt{memory}\ a))\\ \ \wedge\ (\forall d.\ \texttt{OWN}\ i\ d\ \vee \texttt{INTERRUPT}\ d\\ \quad \Rightarrow (s_1.\texttt{devices}.d = s_2.\texttt{devices}.d)\ \wedge\ (s_1.\texttt{ext_out}\ d = s_2.\texttt{ext_out}\ d))\\ \ \wedge\ (s_1.\texttt{psrs}(\texttt{CPSR}) = s_2.\texttt{psrs}(\texttt{CPSR}))\ \wedge\ (s_1.\texttt{coproc.state} = s_2.\texttt{coproc.state})\\ \ \wedge\ (\texttt{nav}\ s_1 = \texttt{nav}\ s_2) \wedge\ (s_1.\texttt{misc} = s_2.\texttt{misc})\end{array}$$

Non-infiltration guarantees that system components outside the `bisim` relation can not give rise to information flow. In particular, privileged registers, memory foreign to the current process and devices that operate on such memory can not influence the execution on the CPU. External output has no impact on other components either. However, it was included into the relation to obtain guarantees on that information from the kernel and neighboring processes can not be leaked through the system's output, as long as the configuration of the devices producing that output prevents them from accessing confidential memory.

6.3 Extended Non-exfiltration

Non-exfiltration guarantees the integrity of resources foreign to the active user process. Given a valid configuration for user process i active, the execution of a single instruction in user mode will not modify any other resources but those considered to be modifiable by i. In [10] this was expressed by the equality of protected components in pre- and poststate. However, when some of those protected components are modified by devices executing in parallel, this equality can not be proven. Therefore, we extend non-exfiltration to a triangle shaped property (compare Fig. 4.b), in which the poststate t of a system-wide progress is compared to both, the prestate s and a third state of comparison r that is the result of applying only the effects of the device operations to the prestate.

Theorem 2. *Extended Non-exfiltration*

$$\begin{aligned}&\forall s,\ t,\ r,\ i.\ (\texttt{mode}\ s = \texttt{usr})\ \wedge\ \texttt{mmu_setup}\ i\ s\ \wedge\ \texttt{device_setup}\ i\ s\\ &\quad\wedge\ (\texttt{next}\ s = \texttt{ValueState}\ ()\ t)\ \wedge\ (\texttt{advance all}\ t.\texttt{counter}\ s = \texttt{ValueState}\ ()\ r)\\ &\quad\Rightarrow \texttt{intact}\ i\ s\ t\ r\end{aligned}$$

For synchronization, `advance` is applied up to the oracle counter state in poststate t. The `intact` relation between the prestate s with active process i, the poststate t and the comparison state r guarantees that coprocessors and memory not belonging to any user process remain unchanged. The memory of neighboring user processes, new interrupts, and devices that do not access memory of i, are determined by the device operations only. In particular, they can not be influenced by writing to the memory of i. The only modifiable registers are the `CPSR`, user mode registers, and the `PSR` and the link register of the mode in t.

$$\begin{aligned}&\texttt{intact}\ i\ s\ t\ r :=\\ &\quad (t.\texttt{coproc} = s.\texttt{coproc})\ \wedge\ (\forall a.(\forall j.\neg\texttt{accessible}\ j\ a) \Rightarrow (t.\texttt{memory}\ a = s.\texttt{memory}\ a))\\ &\wedge (\forall a,\ j.\ (i \neq j)\ \wedge \texttt{accessible}\ j\ a \Rightarrow (t.\texttt{memory}\ a = r.\texttt{memory}\ a))\\ &\wedge (t.\texttt{int_fired} = r.\texttt{int_fired})\\ &\wedge (\forall d.\ \texttt{FOREIGN}\ i\ d\ \vee \texttt{INTERRUPT}\ d\\ &\qquad \Rightarrow (t.\texttt{devices}.d = r.\texttt{devices}.d)\ \wedge\ (t.\texttt{ext_out}\ d = r.\texttt{ext_out}\ d))\\ &\wedge (\forall q.\ q \notin \texttt{accessible_regs}\ (\texttt{mode}\ t) \Rightarrow (t.\texttt{regs}(q) = s.\texttt{regs}(q)))\\ &\wedge (\forall p.\ p \notin \{\texttt{CPSR}, \texttt{spsr_}(\texttt{mode}\ t)\} \Rightarrow (t.\texttt{psrs}(p) = s.\texttt{psrs}(p)))\end{aligned}$$

6.4 Filtered Device Non-Infiltration

In addition to the non-infiltration property of the overall system, we provide one for device activities only. It can be combined with extended non-exfiltration to guarantee that devices not accessing the active partition form their own group of resources which executes independently from the CPU. Formally, filtered device non-infiltration (Fig. 4.c) states that devices configured to not access more than the memory of active process i (devices d for which `OWN` i d holds) cannot influence devices not operating on that memory. Consequently, when comparing two systems and their executions, removing the activities of devices in the `OWN` class from one of the executions (through the filtering predicate of `advance`) will not change the effects that the other devices can observe.

Theorem 3. *Filtered Device Non-Infiltration*

$$
\begin{aligned}
& f_2 = (\lambda d.\ f_1\ d \wedge\ \neg \texttt{OWN}\ i\ d)\ \wedge\ \texttt{devsim}\ i\ s_1\ s_2 \\
\wedge\ & \texttt{device_setup}\ i\ s_1\ \ \wedge\ \ (\texttt{advance}\ f_1\ n\ s = \texttt{ValueState}\ ()\ t_1) \\
\wedge\ & \texttt{device_setup}\ i\ s_2\ \ \wedge\ \ (\texttt{advance}\ f_2\ n\ s = \texttt{ValueState}\ ()\ t_2) \\
\Rightarrow\ & \texttt{devsim}\ i\ t_1\ t_2
\end{aligned}
$$

The `devsim` equivalence relation describes the resources visible to interrupt devices and to devices that operate on memory of non-active user processes.

$$
\begin{aligned}
& \texttt{devsim}\ i\ s_1\ s_2 := \\
& \quad (s_1.\texttt{counter} = s_2.\texttt{counter})\ \wedge\ (s_1.\texttt{int_fired} = s_2.\texttt{int_fired}) \\
& \wedge\ (\forall a,\ j.\ (i \neq j)\ \wedge\ \texttt{accessible}\ j\ a \Rightarrow (s_1.\texttt{memory}\ a = s_2.\texttt{memory}\ a)) \\
& \wedge\ (\forall d.\ \neg \texttt{OWN}\ i\ d \Rightarrow (s_1.\texttt{devices}.d = s_2.\texttt{devices}.d) \wedge (s_1.\texttt{ext_out}\ d = s_2.\texttt{ext_out}\ d))
\end{aligned}
$$

7 Implementation

We proved the theorems of Section 6 for the ARMv7 platform inside HOL4. This work extends the proof presented in [10], in which we showed non-infiltration, non-exfiltration and mode switching properties for ARMv7 user mode execution on ISA level without devices. Given the complexity of the ARM model and the instruction set, we exploited automation based on a sound, but incomplete inference system. For example, for two computations f and g that both preserve non-infiltration, the inference rule for sequential composition derives that also $f \gg_{\texttt{nav}} g$ preserves non-infiltration. We have proven further rules for parallel composition, loops, alternatives, lambda abstraction and other constructors of the operational semantics. They enabled us to develop a proof tool for relational and invariant reasoning that - after being provided with the desired properties for primitive operations - was able to discharge large parts of the proof obligations (but not all) automatically. Details are discussed in [10].

In the present extended work, the separation properties had to be proven manually for `advance`, mainly because they would not hold for intermediate computations in isolation. Due to the complexity, this was one of the main challenges. We followed a bottom-up approach. Basic properties on `mem_acc_by_dev`, a rather extensive case analysis and automatic simplification allowed for the verification of properties on `advance_single`. This step often required to split the analysis into the effects on the device currently progressed by `advance_single` and the effects on all other devices. Finally, properties for `advance` were proven by induction. In order to allow for the continued application of the proof tool to the existing parts, we had to verify the transitivity of `advance`. Subsequently, the vast majority of the automatic proofs could be repeated without any interruptions, which gives confidence that our proof framework scales well for extensions of the platform model. The Cambridge model of ARM is 9 kLOC. In addition to the ARM model, we rely mainly on the relatively small inference kernel of the HOL4 theorem prover, our MMU extension (about 180 lines of definitions), the

device framework (about 350 lines) and the formulation of the discussed properties (about 380 lines). The entire proof script has a length of about 20 kLOC and needs roughly two and a half hours to run on an Intel(R) Xeon(R) X3470 core at 2.9 GHz. We invested about five person months of effort into this work.

8 Conclusion and Future Work

We extended the Cambridge HOL4 ISA model for ARM by a general device framework for DMA devices. Based on the extended model we identified secure device configurations and proved several isolation properties for platforms where DMA devices execute in parallel with a CPU in user mode. The results can be used in separation proofs, be it in a hypervisor, separation kernel or operating system setting. Model, properties and verification approach can be adapted to other architectures. We gained confidence that our proof framework scales well for extensions of the model. The model allows for further interesting angles, which we plan to explore in future work: It is rather common that devices communicate with each other. So far, we can only support such constellations by merging communicating devices into one block, so that the model understands the block as a single device. Removing this restriction comes with the challenge of ensuring that device configurations still remain secure when devices are allowed to write to ports of other devices. Probably easier to achieve is the augmentation of the set of device classes by devices that neither use DMA nor interrupts, but that can be accessed by user space processes. A UART interface managed by a single process is one such example. From a security perspective, such a device is similar to physical memory assigned to a process, in spite of the self-modifying nature and external influence that such components have. Even if devices (like a timer) are shared between different user processes, user mode access to their ports can still preserve isolation, for example, if that access is always reading. Further potential future work includes the investigation of connected external input/output channels or the enhancement of the model by an IOMMU.

Acknowledgments. Work supported by framework grant "IT 2010" from the Swedish Foundation for Strategic Research.

References

1. E. Alkassar and M. A. Hillebrand. Formal functional verification of device drivers. In *Verified Software: Theories, Tools, Experiments*, 2008.
2. E. Alkassar, W. J. Paul, A. Starostin, and A. Tsyban. Pervasive verification of an OS microkernel. In *Verified Software: Theories, Tools, Experiments*, 2010.
3. J. Duan. *Formal verification of device drivers in embedded systems.* PhD thesis, the University of Utah, 2013.
4. J. Duan and J. Regehr. Correctness proofs for device drivers in embedded systems. In *Proceedings of the 5th international conference on Systems software verification*, SSV'10, Berkeley, CA, USA, 2010. USENIX Association.

5. L. Duflot, D. Etiemble, and O. Grumelard. Using CPU system management mode to circumvent operating system security functions. In *Proc. CanSecWest*, 2006.
6. A. C. J. Fox. Formal specification and verification of ARM6. In *Theorem Proving in Higher Order Logics*. Springer Berlin Heidelberg, 2003.
7. A. C. J. Fox and M. O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *Interactive Theorem Proving (ITP)*, 2010.
8. 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, 2008.
9. M. A. Hillebrand, T. In der Rieden, and W. J. Paul. Dealing with I/O devices in the context of pervasive system verification. In *International Conference on Computer Design (ICCD): VLSI in Computers and Processors*, pages 309–316, 2005.
10. N. Khakpour, O. Schwarz, and M. Dam. Machine assisted proof of ARMv7 instruction level isolation properties. In *Certified Programs and Proofs (CPP)*, 2013.
11. D. Monniaux. Verification of device drivers and intelligent controllers: a case study. In *Embedded Software*, 2007.
12. 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 *Security and Privacy*, 2013.