Trustworthy isolation of DMA devices

Jonas Haglund¹ · Roberto Guanciale¹

Received: 6 March 2020 / Accepted: 15 April 2020 / Published online: 15 May 2020
© The Author(s) 2020

Abstract

We present a mechanism to trustworthy isolate I/O devices with direct memory access (DMA), which ensures that an isolated I/O device cannot access sensitive memory regions. As a demonstrating platform, we use the network interface controller (NIC) of an embedded system. We develop a run-time monitor that forces NIC reconfigurations, defined by untrusted software, to satisfy a security rule. We formalized the NIC in the HOL4 interactive theorem prover and we verified the design of the isolation mechanism. The verification is based on an invariant that is proved to be preserved by all NIC operations and that ensures that all memory accesses address allowed memory regions only. We demonstrate our approach by extending an existing Virtual Machine Introspection (VMI) with the monitor. The resulting platform prevents code injection in a connected and untrusted Linux.

Keywords Formal verification · System security · Network interface controller

1 Introduction

Formally verified execution platforms (microkernels [10], hypervisors [11] and separation kernels [6]) constitute key software infrastructure for implementing secure IoT devices. By guaranteeing memory isolation and controlling communication between software components, they prevent faults of non-critical software (e.g., HTTP interfaces, optimizations based on machine learning, and software providing complex functionality or with short life cycle) from affecting software that must fulfill strict security and safety requirements. This enables verification of critical software without considering untrusted software.

A problem with these platforms is that the verification does not consider I/O devices with direct memory access (DMA). Current systems either disable them, use a special System or Input/Output MMU (SMMU or IOMMU; usually unavailable in embedded systems) to isolate potentially

misconfigured devices, or trust the (usually large) controlling software.

In order to address this issue we advocate designing secure IoT and embedded systems using component isolation and the principle of complete mediation: The reconfigurations of the I/O device defined by a device driver are checked by a secure monitor to enable the device to only access certain memory regions. This monitor preserves a security policy which is described by an invariant. The rationale is that the monitor is substantially simpler, and therefore easier to analyze and verify, than the untrusted software. In this context, security depends mainly on three properties: (1) the security policy (i.e., the invariant) implies that the I/O device cannot violate memory isolation; (2) the monitor is correctly isolated from the other, possibly corrupted, components of the system (i.e., the execution platform is formally verified or vulnerabilities are unlikely due to the small code base of the kernel); (3) the monitor is functionally correct and denies configurations that violate the invariant (i.e., the monitor is verified or its small code minimizes the number of critical bugs).

We contribute with the first formal verification of (1) for a real I/O device of significant complexity. As a demonstrating platform we use the embedded system Beaglebone Black (a commonly available development board) and its Network Interface Controller (NIC). We provide a formal model of the NIC and we define the security policy as an invariant

The HOL4 proofs and the source code of the monitor are published at <https://github.com/kth-step/NIC-formalization-monitor>.

✉ Jonas Haglund
jhagl@kth.se
Roberto Guanciale
robertog@kth.se

¹ KTH Royal Institute of Technology, Stockholm, Sweden

in terms of the state of the NIC. We then demonstrate that this policy is sound: The invariant is preserved by the NIC and it restricts memory accesses to predetermined memory regions. The analysis is machine-checked by means of the interactive theorem prover HOL4, which makes our reasoning trustworthy.

To demonstrate the applicability of this approach we implemented a secure connected system. Real systems often: need complex network stacks and application frameworks, have short time to market, require support of legacy features, and adopt binary blobs. For these reasons many applications are dependent on commodity OSs. Our goal is to provide a system that satisfies some desired security properties (e.g., absence of malware), even if the commodity software is completely compromised. We extend the prosper hypervisor [6], which has been previously verified to guarantee property (2), with secure support for network connectivity by deploying a NIC monitor, and analyze the correctness (i.e., property (3)) of the monitor.

The paper is organized as follows. In Sect. 2 we provide a high level description of common DMA controllers in order to demonstrate that the majority of devices are configured similarly to the NIC under analysis. Section 3 presents the security threats posed by the untrusted and potentially compromised device driver of the NIC. The following four Sects. 4–7 describe the contributions for verification of property (1). Section 4 introduces the hardware platform and its formal model, Sect. 5 discusses the correctness of the NIC model, Sect. 6 describes the invariant and the structure of the corresponding proof, and Sect. 7 describes the implementation of the model and proof in HOL4. The next four Sects. 8–11 describe a secure connected system implemented with our design approach. Section 8 presents the extension of the existing hypervisor with the NIC monitor and evaluates the resulting overhead, Sect. 9 describes the monitor, Sect. 10 motivates the correctness of the monitor, and Sect. 11 describes an application of the resulting software platform, which supports remote software upgrade and prevents code injection in a connected (and potentially vulnerable) Linux system. Finally, Sects. 12 and 13 present related work and concluding remarks.

2 DMA controllers

We briefly summarize the main traits of DMA controllers (DMAC). These are hardware modules that offload the CPU by performing transfers between memory and I/O devices. From a security point of view, it is important to restrict the memory accesses performed by DMACs to certain memory regions, since unrestricted accesses can overwrite or disclose code and sensitive data. DMACs can be standalone

hardware modules, or embedded in I/O devices such as NICs and USBs.

There are three common interfaces to configure DMACs (we reviewed 27 DMACs, including NICs, USBs, and standalone DMACs from twelve vendors: ARM, Intel, and Texas Instruments, among others). In the simplest DMACs (3 USBs), source, destination, and size of memory buffers to transfer are configured via dedicated registers of the controller.

The seemingly most common method for configuring DMACs (22 controllers of all kinds) is by means of linked lists of Buffer Descriptors (BD), an example of which is given in Fig. 1. The list is stored in memory, where BDs specify source (buf1–buf3) and destination buffers (buf4–buf6) by means of pointers and sizes. DMA transfers are activated by writing the address of the head of the list to a specific DMAC register. The DMAC then processes the list in order. First, the current BD is fetched to its local memory. Then a number of bytes are read from the source buffer and written to the destination buffer via local memory. This step is repeated until all bytes of the buffer have been transferred, at which point the complete bit is set to signal that the transfer is complete. The DMAC then processes the next BD, which is addressed by the next descriptor pointer. This procedure continues until the DMAC reaches the end of the list. In this example, the first BD has been processed and the DMAC is currently processing the second BD.

Finally, some DMACs (2 standalone DMACs) are programmable: A program is stored in memory, and which is subsequently fetched and executed by the DMAC to perform the specified memory transfers.

3 Security threats, challenges, and scope

The main concern when DMACs are controlled by untrusted software is that the destination addresses of BDs can be set arbitrarily. Therefore a malicious software could use the DMAC to inject code and data into the execution platform (e.g., hypervisor) or other components (e.g., other guests), or modify page tables and escalate its privileges. Similarly, by

Fig. 1 An example of a DMAC that performs memory-to-memory transfers and that is configured via linked list of BDs

controlling the source addresses of BDs, an attacker can use the DMAC to leak arbitrary regions of memory. Finally, the untrusted software may configure a DMAC in ways that do not follow the specification, causing the system to perform unknown operations.

Easy protection against these threats is to completely isolate the DMAC from sensitive memory regions via an SMMU/IOMMU. This is a hardware component that restricts the memory accesses of I/O devices. Unfortunately, even capable embedded systems do not have an SMMU. Moreover, an SMMU may have negative impacts on cost, performance, and power consumption, and introduce I/O jitter.

Software prevention against these threats requires a detailed understanding of the DMAC. We address this by defining a detailed and unambiguous mathematical model of the DMAC under analysis: The NIC of BeagleBone Black. The challenge with defining such a model is to understand the NIC specification, which is ambiguous, dispersive, self-contradictory and vague, and contains many details that are not security relevant.

An additional challenge is the identification of the security invariant, presented in Sect. 6.1. Each transition of the NIC model (c.f. Sect. 4) describes a small set of operations, leading to a complicated state with many components. Also, the NIC writes BDs after transmission and reception of frames. Many of these state components, and all these writes must be considered when defining the invariant. For instance, if the NIC writes a BD that overlaps another BD, then the destination address of the other BD might be modified. All these details make the formal verification challenging, but helped us discover bugs in the Linux NIC device driver, errors in the NIC specification, and define a monitor policy that includes security relevant details that may otherwise be overlooked. These findings are summarized in Sect. 13.

We remark that our goal is to define and verify an invariant of the NIC that can be used to define the security policy of the NIC monitor. Verifying that the NIC hardware implements its specification is not considered.

4 Hardware platform and formal model

Our analysis concerns the development board BeagleBone Black. We take into account only the Ethernet NIC and assume that the other DMACs of the SoC (e.g., USB) are disabled.

Our formal model of the SoC uses the device model framework by Schwarz et al. [15], which describes executions of computers consisting of one ARMv7 CPU, memory and a number of I/O devices. The state of the CPU-memory subsystem [8] is represented by a pair $s = (c, m)$, where c is

a record representing the contents of the CPU registers, and m is a function from 32-bit words to 8-bit bytes representing the memory.

The state of the NIC is described by a record $n = (reg, it, tx, rx, td, rd)$. The first component describes the interface between the CPU and the NIC, consisting of the memory-mapped NIC registers: Ten 32-bit registers $reg.r$ and an 8-kB memory $reg.BD_RAM$. The other components (it, tx, rx, td, rd), describe the internal state of the NIC, consisting of five records representing components of five automata. Each automaton describes the behavior of one of the five NIC functions: *initialization* (it), *transmission* (tx) and *reception* (rx) of frames, and *tear down of transmission* (td) and *reception* (rd).

A NIC transition enters an undefined state $\perp$ if any of the following conditions hold: (1) the transition results due to a NIC register write that does not follow the NIC specification [1] (e.g., that a specific register should be written with a specific value when the NIC is in a specific state), (2) the transition occurs from a state from which operations are not described by the specification (e.g., the result of issuing a DMA request not addressing RAM), or (3) the operation is not included in the formal NIC model (e.g., not relevant for memory accesses).

The execution of the system is described by a transition relation $(s, n) \rightarrow (s', n')$, which is the smallest relation satisfying the following rules

$$\begin{array}{c} \frac{s \xrightarrow{\tau_{cpu}} s'}{(s, n) \rightarrow (s', n)} \quad \frac{s \xrightarrow{req(a)} s'' \quad s'' \xrightarrow{rep(a, n.reg[a])} s'}{(s, n) \rightarrow (s', n)} \\[10pt] \frac{s \xrightarrow{write(a, v)} s' \quad n \xrightarrow{update(a, v)} n'}{(s, n) \rightarrow (s', n')} \\[10pt] \frac{n \xrightarrow{\tau_{atm}} n'}{(s, n) \rightarrow (s, n')} \quad \frac{n \xrightarrow{write(a, v)} n'}{(c, m, n) \rightarrow (c, m[a := v], n')} \\[10pt] \frac{n' \xrightarrow{req(a)} n'' \quad n'' \xrightarrow{rep(a, m[a])} n'}{(c, m, n) \rightarrow (c, m, n')} \end{array}$$

where $s \xrightarrow{l} s'$ and $n \xrightarrow{l} n'$ denote the transition relations of the CPU-memory subsystem and the NIC, respectively. Notice that these rules are general enough to handle other types of DMACs. To include fine-grained interleavings of the operations of the CPU and the NIC, each NIC transition describes one single observable hardware operation: One register read or write, one BD read, one BD field write, or one single memory access of one byte.

The first two rules do not affect the NIC: the CPU can execute an instruction that (1) does not access a memory mapped NIC register ($s \xrightarrow{\tau_{cpu}} s'$), or (2) that reads the NIC

register at address a ($s \xrightarrow{req(a)} s''$) and processes the result ($s'' \xrightarrow{rep(a, n.reg[a])} s'$). The third rule describes executions of CPU instructions writing a value v to the NIC register at address a ($s \xrightarrow{write(a, v)} s'$). Register writes configure the NIC and may activate an automaton ($n \xrightarrow{update(a, v)} n'$).

The other three rules involve transitions of active automata. An internal transition of an automaton $atm \in \{it, tx, rx, td, rd\}$ ($n \xrightarrow{\tau_{atm}} n'$) does not affect the CPU. Memory write requests of writing a byte value v to a location with the address a ($n \xrightarrow{write(a, v)} n'$) are issued only by the transmission automaton tx . Memory read requests of reading the memory byte at an address a ($n' \xrightarrow{req(a)} n''$) are issued only by the reception automaton rx , and the byte value at the addressed memory location ($m[a]$) is immediately processed by the NIC ($n'' \xrightarrow{rep(a, m[a])} n'$).

The remainder of this section describes the five automata.

Initialization

Figure 2 depicts the initialization automaton. Initially, the automaton is in the state *power_on* ($n.it.s = power_on$). Initialization is activated by writing 1 to the register RESET ($n.reg.RESET$), causing the automaton to transition to the state *reset*. The transition from *reset* to *idle* is inhibited until the transmission and reception automaton have reach the *idle* state. When the automaton reach the transitions to the state *init_regs* it sets RESET to 0. The CPU completes the initialization by clearing the transmission and reception HDP and CP registers (explained below), causing the automaton to enter the state *idle*. The NIC can now be used to transmit and receive frames. If the CPU does not initialize registers as described, then the NIC enters $\perp$ (i.e., $n' = \perp$), since any other behavior is unspecified.

Transmission and reception

The NIC is configured via linked lists of BDs. One frame to transmit (receive) can be stored in several buffers scattered in memory, the concatenation of which forms the frame. The properties of a frame and the associated buffers

Fig. 2 Initialization automaton: r is the address of the RESET register. p ranges over the addresses of the HDP and CP registers that must be cleared to complete the initialization of the NIC

are described by a 16-byte BD. In contrast to the example illustrated in Fig. 1, the lists of BDs are located in the internal NIC memory ($n.reg.BD_RAM$). There is one queue (linked list) for transmission and one for reception, which are traversed by the NIC during transmission and reception of frames. Each BD contains among others the following fields: Buffer Pointer (BP) identifies the start address of the associated buffer in memory; Buffer Length (BL) identifies the byte size of the buffer; Next Descriptor Pointer (NDP) identifies the start address of the next BD in the queue (or 1 if the BD is last in the queue); Start/End Of Packet (SOP/EOP) indicates whether the BD addresses the first/last buffer of the associated frame; Ownership (OWN) specifies that the NIC has not completed the processing of the BD; End Of Queue (EOQ) indicates whether the NIC considered the BD to be last in the queue when the NIC processed that BD (i.e., NDP was equal to 0). Figure 3 shows an example of a BD queue.

The state of the transmission automaton consists of the following fields:

- $n.tx.s$: The state of the transmission automaton.
- $n.tx.bda$: The address of the currently processed BD (can be a SOP, EOP or neither), or 0 if no BD is currently processed.
- $n.tx.start-bda$: The address of the first BD in the transmission queue (always a SOP), or 0 if no queue is currently processed.
- $n.tx.eop-bda$: The address of the EOP BD currently processed.
- $n.tx.bd$: A record containing values of the fields of the currently processed BD.
- $n.tx.mem_a$: The address of the next byte of the frame to transmit.
- $n.tx.bytes_left$: The number of bytes of the frame to transmit that are left.

Fig. 3 A buffer descriptor queue consisting of three BDs located in the memory of the NIC. The queue starts with the topmost BD, which addresses the first buffer of the first frame (SOP = 1) and is linked to the middle BD. The middle BD addresses the last (and second) buffer of the first frame (EOP = 1) and is linked to the bottom BD. The bottom BD is last in the queue (NDP = 0) and addresses the only buffer of the second frame (SOP = EOP = 1)

The initial state of the transmission automaton (Fig. 4) is $n.tx.s = idle$. The CPU activates transmission by writing the transmission head descriptor pointer register ($n.reg.TX_HDP$) with the address of the first BD in the queue addressing the frames to transmit. Such a NIC register write causes $n'.tx.bda$ and $n'.tx.start-bda$ to contain the written address, and the next state to be $n'.tx.s = fetch_bd$. If TX_HDP is written when it is not 0, or when transmission teardown is active, then $n' = \perp$.

The transition from $fetch_bd$ reads $n.reg.BD_RAM$ to decode the fields of the current BD located at $n.tx.bda$ and sets $n'.tx.bd$ to the values of the read fields. If the BD is well-formed, then $n'.tx.mem_a$ and $n'.tx.bytes_left$ are set to values computed from fields of the read BD, and the resulting automaton state is mem_req . If the BD is not well-formed (e.g., the fetched BD is outside BD_RAM or its location is not 4-byte aligned, or certain BD fields are not properly initialized), then the NIC enters $\perp$.

As long as there are bytes of the buffer left to read and transmit ($n.tx.s = mem_rep \wedge n.tx.bytes_left > 0$ or $n.tx.s = mem_req$), the automaton transitions between mem_req and mem_rep , fetching and processing in each cycle one byte from memory via DMA. The transition from mem_rep that processes the last byte of the buffer addressed by the current BD ($n.tx.bytes_left = 0$) enters either $fetch_bd$ or eoq_own . The transition to $fetch_bd$ is performed if the currently transmitted frame consists of additional buffers that need to be fetched from memory (i.e., the EOP flag is not set of the current BD: $n.tx.bd.eop = false$), and which sets the address of the current BD to the address of the next BD ($n'.tx.bda = n.tx.bd.ndp$). eoq_own is entered if the all bytes of the current frame has been fetched from memory (i.e., the EOP flag is set of the current BD: $n.tx.bd.eop = true$), and which saves the address of the current (EOP) BD ($n'.tx.eop-bda = n.tx.bda$; this assignment is made since the value $n.tx.bda$ is needed by the later transitions from $write_cp$ to $idle$ and $fetch_bd$, but $n.tx.bda$ is overwritten by either the transition from eoq_own to $write_cp$ or by the transition from own_hdp to $write_cp$).

Fig. 4 Transmission automaton: $tx-a$ is the address of the transmission head descriptor pointer register, which is written to trigger transmission of the frames addressed by the BDs in the queue whose head is at $bd-a$. The address $mem-a$ is the memory location requested to read, and v is the byte value in memory at that location

Once in the state eoq_own , if the current BD is not last in the queue (i.e., the NDP field of the current BD is not 0: $n.tx.bd.ndp \neq 0$), then the OWN flag is cleared in BD_RAM ($n'.reg.BD_RAM$) of the SOP BD at $n.tx.start-bda$ (indicating to a device driver that the memory area in BD_RAM of the BDs of the transmitted frame can be reused), and $n'.tx.bda$ and $n'.tx.start-bda$ are set to the address of the next BD ($n'.tx.bda = n'.tx.start-bda = n.tx.bd.ndp$; identifying the next BD to process and advancing the transmission queue to start from the next BD, respectively), and enters the state $write_cp$. If the current BD is last in the queue (i.e., the NDP field of the current BD is 0: $n.tx.bd.ndp = 0$) then the EOQ flag is cleared in $n'.reg.BD_RAM$ of the current BD located at $n'.tx.bda$ (used by a device driver to check whether a BD was appended just after the NIC processed a BD, which would result in the NIC not processing the appended BD, meaning that a device driver must restart transmission) and enters the state own_hdp . The transition from own_hdp clears the OWN flag in $n'.reg.BD_RAM$ of the SOP BD at $n.tx.start-bda$, TX_HDP ($n'.reg.TX_HDP = 0$; $TX_HDP = 0$ indicates to a device driver that all frames have been transmitted), and sets $n'.tx.bda = 0$ and $n'.tx.start-bda = 0$ (indicating that there is no current BD to process and that the transmission queue is empty).

The transition from $write_cp$ writes the address of the just processed (EOP) BD to the transmission completion pointer register ($n'.reg.TX_CP = n.tx.eop-bda$) to inform a device driver of which is the last processed BD (this raises a frame transmission completion interrupt which a device driver can acknowledge by writing TX_CP with the address of the last processed BD). Furthermore, if all BDs in the BD queue have now been processed ($n.tx.bda = 0$), or initialization or transmission teardown was requested during the processing of the BDs of the last transmitted frame ($n.it.s \neq idle \vee n.td.s \neq idle$), then the next state is $idle$. Otherwise the next state is $fetch_bd$ to begin the processing of the first BD of the next frame.

The structure of the reception automaton is similar to the structure of the transmission automaton but with four notable differences: (1) after the reception head descriptor pointer has been written with a BD address to enable reception, it is non-deterministically decided when a frame is received to activate the reception automaton. (2) The BDs in the reception queue address the buffers used to store received frames. Since reception do not get memory read replies there is only one state related to memory accesses. (3) The transmission automaton has two states (eoq_own and own_hdp) to describe BD writes (of the flags EOQ and OWN). Reception writes sixteen BD fields (e.g., the length of a frame and the result of a CRC check), leading to fourteen additional states. (4) Since content of received frames are unknown, values written to memory and some BD fields are selected non-deterministically.

Tear down

The initial state of the transmission teardown automaton (Fig. 5) is $n.td.s = idle$. When the CPU writes 0 to the transmission teardown register ($n.reg.TX_TD$), the state $set-eq$ is entered. However, if TX_TD is written when the NIC is not initialized ($n.it.s \neq idle$), transmission teardown is in progress ($n.td.s \neq idle$), or a non-zero value is written, then the NIC enters $\perp$.

Before a transition can be performed from $set-eq$, the transmission automaton must first complete the processing of the currently transmitted frame ($n.tx.s = idle$). Then there are two cases depending on whether all BDs in the transmission queue were processed or not. If all BDs were processed ($n.tx.bda = 0$), then the transition from $set-eq$ to $write-cp$ is performed, clearing TX_HDP ($n'.reg.TX_HDP = 0$) and $n'.tx.bda = 0$. Otherwise ($n.tx.bda \neq 0$), there are two non-deterministic cases of either setting the EOQ flag (the transition to $set-td$) or the teardown flag (the transition to $own-hdp$) of the BD (at address $n.tx.bda$) that follows the last processed BD. The reason for this non-deterministic behavior is because the NIC specification does not state that this operation is performed but tests on the hardware shows that this is indeed the cases, making the model cover both cases. The transition from $set-td$ clears the teardown flag of the BD at $n.tx.bda$. The transition from $own-hdp$ clears TX_HDP the OWN flag of the BD at $n.tx.bda$, $n'.tx.bda = 0$ and $n'.tx.start-bda = 0$. Finally, the transition from $write-cp$ writes the teardown completion code $0 \times FFFFFFFC$ to $n'.reg.TX_CP$ to signal to the CPU that the teardown is complete.

Reception teardown works in a similar way but has two more states for writing additional BD fields.

5 Model validation

This section considers the correctness of the NIC model. There are three components that affect correctness: The model, the specification and the hardware implementation of the NIC. If any of these components do not describe the

intended behavior, then the NIC model is most likely incorrect. For instance, there could be a typo, logical error or ambiguity in the model or specification, or there could be an error in the hardware.

To minimize inconsistency between the model and the specification, both the model and the specification have been reviewed several times. In addition, we studied the Linux NIC driver to clarify vague statements in the specification. Still, there are some unknowns. An example of an inconsistency in the specification is: One section states that a certain BD flag is set of the SOP BD while another section states that the flag is set in the EOP BD. To include all possibilities, the model is non-deterministic, causing it to either set the flag in the SOP BD, in the EOP BD or in both.

The effects of some operations are not completely specified. For instance, the specification states that TX_HDP becomes 0 after the complete transmission queue has been processed, but nothing is stated about how TX_HDP changes its value during transmission. The model describes this behavior by setting TX_HDP to a non-deterministic value distinct from 0. The internal state component $n.tx.start-bda$, not accessible to the CPU, is therefore introduced to record the head of the queue. If TX_HDP always contains the address of the head of the queue, then this non-determinism and the state component $n.tx.start-bda$ would not be needed.

Moreover, the specification includes instructions of how the NIC should be configured. For instance, TX_HDP shall be written with the address of the first BD of a queue to be processed for transmission, but TX_HDP should not be written when not 0. The NIC model enters an undefined state (i.e., $\perp$) when these instructions are not followed. The model also enters an undefined state when operations shall be performed that are unspecified or unclear. For example, the specification does not state the effect of activating transmission while transmission teardown is in progress.

To minimize inconsistency between the model and the actual hardware, the NIC has been tested to observe how the NIC updates its registers and BD fields. For example we discovered that the NIC sets the EOQ flag of the first unprocessed BD during teardown, which is unstated in the specification. This behavior is described non-deterministically by the model (the transitions from $set-eq$). In order to not inadvertently omit possible interleaving between NIC and CPU operations, the NIC transitions are fine-grained: each NIC transition describes a single BD field write or memory byte access. Finally, the verification exercised the model by means of a significant number of lemmas (e.g., the queues shrink during transmission and reception).

Despite this conservative definition of the model, some inconsistencies were found. The teardown automata does not check whether the BD to write is in BD_RAM . For our verification, this error is not critical since the NIC invariant (c.f. Sect. 6.1) guarantees that every BD is in BD_RAM . We

Fig. 5 Transmission teardown automaton: $td-add$ is the address of TX_TD which is written with 0 to trigger teardown of transmission

also identified an inconsistency by analyzing a simplified model of transmission with the NuSMV [4] model checker (a model with a small address space and few BD fields to make the analysis feasible). The order of the operations of transmission differs from the order that can be inferred, via non-trivial reasoning, from the specification. This inconsistency and non-trivial reasoning illustrate the challenge of manual modeling based on informal specifications. This error is also not critical for the verification of the invariant of Sect. 6 since it only affects the order of transitions. The error may affect the formal verification of the NIC monitor, since the order of transitions affect the synchronization between the CPU and the NIC.

6 Formal verification of NIC isolation

Our main verification goal is to identify a NIC configuration (state) that isolates the NIC from certain memory regions. This means that the NIC can only read and write certain memory regions, denoted by R and W respectively. We identify such a configuration by means of an invariant $\mathcal{I}_{NIC}$ that is preserved by internal NIC transitions ($l \neq \text{update}(a, v)$) and that restricts the set of accessed memory locations:

Theorem 1 $\mathcal{I}_{NIC}(n, R, W) \wedge n \xrightarrow{l} n' \wedge l \neq \text{update}(a, v)$ implies

1. $\mathcal{I}_{NIC}(n', R, W)$,
2. $l = \text{req}(a) \implies a \in R$, and
3. $l = \text{write}(a, v) \implies a \in W$.

6.1 Definition of the invariant

In order to facilitate the definition, the invariant of the NIC model is split into several sub-invariants:

$$\mathcal{I}_{NIC}(n, R, W) := \bigwedge_{i \in \{wd, qs, it, tx, rx\}} \mathcal{I}_i(n, R, W)$$

6.1.1 Well-defined state

$\mathcal{I}_{wd}(n, R, W) := n \neq \perp$ states that the NIC is in a defined state. This ensures that the NIC cannot perform unspecified (arbitrary) operations (transitions) that would potentially violate memory isolation.

6.1.2 Disjoint queues

$\mathcal{I}_{qs}$ states that when the transmission and reception automata are active, no BD in the transmission queue overlaps

a BD in the reception queue, and vice versa (no byte in $n.\text{reg.BD_RAM}$ is used by both a BD in the transmission queue and by a BD in the reception queue):

$$\begin{aligned} \mathcal{I}_{qs}(n, R, W) &:= n.\text{tx}.s \neq \text{idle} \wedge n.\text{rx}.s \neq \text{idle} \\ &\implies \text{SEP}(q_{tx}^{nic}(n), q_{rx}^{nic}(n)) \end{aligned}$$

The functions q_{tx}^{nic} and q_{rx}^{nic} denote the list of the addresses of the BDs in the transmission and reception queues of the NIC, respectively: Let $q(n, a)$ denote the list of addresses of the BDs in the queue starting at address a in the state n and $q(n, 0) = []$, then $q_{tx}^{nic}(n) = q(n, n.\text{tx}.start\text{-}bda)$, and $q_{rx}^{nic}(n) = q(n, n.\text{rx}.start\text{-}bda)$.

This invariant ensures that transmission and transmission teardown do not affect nor are affected by the state components of the reception and reception teardown automata, and vice versa. In particular this property guarantees that when the transmission (reception) automaton writes into the transmission queue, it cannot modify the content of the reception (transmission) queue, which would otherwise potentially cause the reception automaton to violate memory isolation.

6.1.3 Initialization

$\mathcal{I}_{it}$ implies that when initialization is complete (the initialization automaton transitions from init_regs to idle), the transmission and reception automata are idle.

$$\begin{aligned} \mathcal{I}_{it}(n, R, W) &:= n.\text{it}.s = \text{init_regs} \\ &\implies n.\text{tx}.s = \text{idle} \wedge n.\text{rx}.s = \text{idle} \end{aligned}$$

Only the transmission and reception automata can perform internal transition that may cause the NIC model to enter an undefined state or access memory. Hence, $\mathcal{I}_{it}$ implies that when initialization is complete, the transmission and reception automata cannot perform such transitions, and that $\mathcal{I}_{tx}$ and $\mathcal{I}_{rx}$ hold vacuously (see below for the definition of $\mathcal{I}_{tx}$).

6.1.4 Transmission

$\mathcal{I}_{tx}$ is split in into two conjuncts:

$$\begin{aligned} \mathcal{I}_{tx}(n, R, W) &:= (n.\text{tx}.s \neq \text{idle} \\ &\implies \mathcal{I}_{tx\text{-}wd}(n) \wedge \mathcal{I}_{tx\text{-}mr}(n, R) \wedge \\ &\quad (n.\text{tx}.s = \text{idle} \wedge n.\text{tx}.bda \neq 0 \\ &\implies \text{SEP}([n.\text{tx}.bda], q_{rx}^{nic}(n))) \end{aligned}$$

The first conjunct applies when the transmission automaton is in a state from which it can perform an internal transition and prevents these transition from entering an undefined state or reading unreadable memory (to prevent the problems mentioned in the first paragraph of Sect. 3).

To prevent the transmission automaton from causing the NIC to enter $\perp$, $\mathcal{I}_{tx-wd}$ consists of a number of constraints, including:

- The location of each BD in the transmission queue has a 4-byte aligned address in BD_RAM.
- Each BD in the transmission queue is properly initialized. For instance, the OWN flag is set and the buffer length field is greater than zero.
- Each transmission BD is both a SOP and an EOP. To prevent the NIC model from entering $\perp$, each SOP BD must have a matching EOP BD. Since Linux configures each transmission BD to be both a SOP and an EOP, this statement is stronger than necessary, but simplifies the proof of that $\mathcal{I}_{tx-wd}$ is preserved.
- The currently processed BD is the head of the transmission queue ($n.tx.bda = n.tx.start-bda$). This statement is an invariant as a consequence of the previous statement of BDs being both SOP and EOP. This is mainly used to simplify the proof.
- No pair of BDs in the transmission queue overlap each other. This prevents the NIC from writing BD fields such that other BDs, processed in the future, get modified. This has two implications with respect to preventing transitions to $\perp$ and reading unreadable memory: the NIC cannot modify (properly initialized) BDs fields that can cause transitions to $\perp$ (e.g., SOP, EOP, buffer length), nor the buffer pointer field (initialized to address readable memory) to address unreadable memory.
- The transmission queue is not circular. If the queue is circular and the transmission automaton modifies fields of a BD, then that modified BD remains in the queue and may be processed again. That modification may cause the BD to violate $\mathcal{I}_{tx}$ (c.f. the second bullet of this list).
- The transmission queue is not empty if the transmission automaton is in a state where a BD is currently being processed or will be processed in the future ($n.tx.s \neq idle \wedge \neg(n.tx.s = write_cp \wedge n.tx.start-bda = 0)$). This is an example of a statement that is included in $\mathcal{I}_{tx}$ for the purpose of proving that $\mathcal{I}_{tx}$ is preserved.

To ensure that the transmission automaton only reads readable memory, $\mathcal{I}_{tx-mr}$ requires that:

- Each BD in the transmission queue addresses the memory region R .
- If the transmission automaton is in the frame fetching loop ($n.tx.s = mem_req \vee n.tx.s = mem_rep$), then the state components used to compute the memory addresses do not cause overflow, and the addresses of future memory read requests issued during the processing of the current BD are in R ($\forall 0 \leq i < n.tx.bytes_left. n.tx.mem_a + i \in R$, where

$n.tx.bytes_left$ records the number of bytes left to read of the buffer addressed by the current BD, and $n.tx.mem_a$ records the address of the next memory read request; see Fig. 4).

The second conjunct of $\mathcal{I}_{tx}$ applies when the transmission automaton is in a state from which it cannot perform an internal transition. In these cases, the reception and the transmission teardown automata may be in states with enabled internal transitions. However, only the reception automaton can perform internal transitions that potentially cause the NIC model to enter an undefined state or writes unwritable memory (the reception automaton does not read memory). Therefore, the transmission teardown automaton must be restricted from affecting the reception automaton. Notice that it is sufficient to restrict the transmission teardown automaton when the transmission automaton is idle, since the former cannot perform transitions when the latter is active. The transmission teardown automaton may affect the reception automaton when it writes fields of the BD at $n.tx.bda$ in BD_RAM, because BD_RAM contains the reception queue. The transmission teardown automaton writes BD_RAM only when $n.tx.bda \neq 0$. Therefore, the second part of $\mathcal{I}_{tx}$ requires that the BD at address $n.tx.bda \neq 0$ is separated from each (does not overlap any) BD in the reception queue (they do not share a byte location in BD_RAM).

6.1.5 Reception

The invariant for reception is similar to the invariant for transmission. The main difference is the definition of $\mathcal{I}_{rx-wd}$, since reception BDs specify different properties than transmission BDs. Also, the invariant states that BDs in the reception queue address buffers in W , and that $n.rx.bda$ is disjoint from the transmission queue.

6.2 Proof of Theorem 1

The proof of Theorem 1.2 and Theorem 1.3 are straightforward. Transitions of the form $n \xrightarrow{req(a)} n'$ occur only when $n.tx.s = mem_req$, where $a = n.tx.mem_a$. $n.tx.s = mem_req \implies n.tx.mem_a \in R$ is implied by $\mathcal{I}_{tx-mr}(n, R)$. Hence, the requested address is readable: $a \in R$. The proof of Theorem 1.3 has the same structure but follows from $\mathcal{I}_{rx}(n, R, W)$.

Defining the invariant in terms of sub-invariants stating properties of initialization, transmission or reception naturally leads the proof of Theorem 1.1 to be described in terms of these three types of actions the NIC performs: $act \in \{it, tx, rx\}$. The labels of the transitions describing one of these three types of actions are identified by $L(act)$, where:

- $L(it) := \{\tau_{it}\}$
- $L(tx) := \{\tau_{tx}, \tau_{td}\} \cup \bigcup_{a,v} \{req(a), rep(a, v)\}$
- $L(rx) := \{\tau_{rx}, \tau_{rd}\} \cup \bigcup_{a,v} \{write(a, v)\}$.

The following two lemmas formalize properties of the NIC model: Transitions of an action do not modify state components of other actions; and an automaton can leave the idle state only when the CPU writes a NIC register.

Lemma 1 For every act , if $n \xrightarrow{l} n'$ and $l \notin L(act)$ then $n'.act = n.act$.

Lemma 2 For every atm , if $n \xrightarrow{l} n'$, $n.atm.s = idle$, and $n'.atm.s \neq idle$ then $l = update(a, v)$.

Lemma 3 states that all transitions of each action, $act \in \{it, tx, rx\}$, preserve the corresponding invariant:

Lemma 3 For every act , if $\mathcal{I}_{NIC}(n, R, W)$, $n \xrightarrow{l} n'$, and $l \in L(act)$ then $\mathcal{I}_{act}(n', R, W)$ and $n' \neq \perp$.

Proof We sketch the proof for $act = tx$, since reception is analogous and initialization is straightforward. The transition l belongs to the transmission or the transmission tear down automaton. There are four cases depending on whether $n.tx.s$ and $n'.tx.s$ are equal to *idle* or not.

Case 1 $n.tx.s = idle \wedge n'.tx.s \neq idle$ cannot occur by Lemma 2.

Case 2 $n.tx.s \neq idle \wedge n'.tx.s \neq idle$ implies that the transition is performed by the transmission automaton ($l = \tau_{tx}$). We first analyze modifications of the transmission queue. The transmission automaton can only modify the flags OWN and EOQ of the currently processed BD (at $n.tx.bda$) and advance the head of the transmission queue ($n'.tx.start-bda = n.tx.bd.ndp$; although not atomically). $\mathcal{I}_{tx-wd}(n)$ implies that the current BD is the head of $q_{tx}^{nic}(n)$ ($q_{tx}^{nic}(n) = [n.tx.bda] \cdot t$ for some possibly empty tail t , where $\cdot$ denotes concatenation) and that the BDs in $q_{tx}^{nic}(n)$ do not overlap. Therefore, the two flag modifications do not alter the NDP fields of the current BD (at $n.tx.bda$) nor the following BDs (at the addresses listed by t) in $q_{tx}^{nic}(n)$. For this reason the transmission queue is only either *unmodified* ($q_{tx}^{nic}(n') = q_{tx}^{nic}(n)$) or *shrunk* ($q_{tx}^{nic}(n') = t$), thereby implying $\mathcal{I}_{tx-wd}(n')$. Moreover, the BP fields are not modified meaning that the buffers addressed by the BDs in $q_{tx}^{nic}(n')$ are still located in R . Therefore $\mathcal{I}_{tx-mr}(n', R)$ holds. The modifications of OWN and EOQ of the current BD do not violate the invariant, since the queue is acyclic, implying that the current BD (at $n.tx.bda$) is not part of the new queue ($q_{tx}^{nic}(n') = t$).

We now analyze modifications of the state components that are used for address calculations of the memory read requests, which are restricted by $\mathcal{I}_{tx-mr}(n, R)$. If the transition

is from *fetch_bd*, then the automaton reads the current BD from $n.reg.BD_RAM$, and assigns the read values to the record $n'.tx.bd$. $\mathcal{I}_{tx-wd}(n)$ implies that the overflow restrictions are satisfied by the fetched BD and hence by the relevant state components in n' , and that the buffer addressed by the fetched BD is in readable memory. These properties are preserved by transitions from *mem_rep* and *mem_req*.

Case 3 $n.tx.s \neq idle \wedge n'.tx.s = idle$. It must be shown that if $n'.tx.bda \neq 0$, then $n'.tx.bda$ does not overlap any BD in $q_{rx}^{nic}(n')$. The only possible transition in this case is made by the transmission automaton when $n.tx.s = write_cp$. Such transitions do not modify $n.tx.bda$, $n.tx.start-bda$, $n.reg.BD_RAM$, nor $n.rx$. Hence, $q_{tx}^{nic}(n') = q_{tx}^{nic}(n)$ and $q_{rx}^{nic}(n') = q_{rx}^{nic}(n)$, which are disjoint by $\mathcal{I}_{qs}(n, R, W)$. Since $n'.tx.start-bda = n'.tx.bda \neq 0$ and $q_{tx}^{nic}(n') = q_{tx}^{nic}(n)$, $n'.tx.bda$ is the first element of $q_{tx}^{nic}(n')$. Hence, $n'.tx.bda$ does not overlap any BD in $q_{rx}^{nic}(n')$.

Case 4 $n.tx.s = idle \wedge n'.tx.s = idle$. These transitions are performed by the transmission tear down automaton ($l = \tau_{td}$), and only write fields of the BD at $n.tx.bda$ (provided $n.tx.bda \neq 0$) and set $n.tx.bda$ to 0. The second conjunct of $\mathcal{I}_{tx}(n, R, W)$ implies that the BD at $n.tx.bda$ does not overlap $q_{rx}^{nic}(n)$, therefore $q_{rx}^{nic}(n) = q_{rx}^{nic}(n')$ and $\mathcal{I}_{tx}(n', R, W)$ holds. $\square$

The following definitions, lemmas and corollaries are used to prove that each action preserves the invariant of other actions and it does not cause the queues to overlap. First, for each action act , we introduce a relation on NIC states, $n \succeq_{act} n'$, with the meaning that the invariant $\mathcal{I}_{act}$ is preserved from n to n' . For initialization, the relation $n \succeq_{it} n'$ states that the state components of the initialization automaton are equal ($n.it = n'.it$) and that the transmission and reception automata remain in their idle states ($\bigwedge_{atm \in \{tx, rx\}} (n.atm.s = idle \implies n'.atm.s = idle)$). For $act \in \{tx, rx\}$, $n \succeq_{act} n'$ states that the:

- state components of the corresponding automaton are equal: $n.act = n'.act$.
- locations of the corresponding queues are equal: $q_{act}^{nic}(n) = q_{act}^{nic}(n')$.
- content of the corresponding queues are equal: $\forall a \in q_{act}^{nic}(n). bd(n, a) = bd(n', a)$, where $\in$ denotes list membership and $bd(n, a)$ is a record with its fields set to the values of the corresponding fields of the BD at address a in the state n .
- other queue is not expanded: $\forall a. a \in q_{act'}^{nic}(n') \implies a \in q_{act'}^{nic}(n)$, where $act' = tx$ if $act = rx$ and $act' = rx$ if $act = tx$.

The following Lemma states that $n \succeq_{act} n'$ indeed preserves the corresponding invariant $\mathcal{I}_{act}$:

Lemma 4 For every act , if $\mathcal{I}_{act}(n, R, W)$ and $n \succeq_{act} n'$ then $\mathcal{I}_{act}(n', R, W)$.

To complete the proof we introduce a relation for every action act , $n \sqsubseteq_{act} n'$, which formalizes that the location of the corresponding queue is unmodified and that all bytes outside the queue are unmodified:

$$\begin{aligned} n \sqsubseteq_{act} n' \\ &:= (\forall a \in q_{act}^{nic}(n). bd(n, a).ndp = bd(n', a).ndp) \wedge \\ &(\forall a \notin \mathcal{A}(q_{act}^{nic}(n)). \\ &n.reg.BD_RAM(a) = n'.reg.BD_RAM(a)) \end{aligned}$$

(where $\mathcal{A}(q_{act}^{nic}(n))$ is the set of byte addresses of the BDs in $q_{act}^{nic}(n)$, and the imaginary “initialization-queue” is defined to be empty: $q_{it}^{nic}(n) := []$). The following Lemma states that each action preserves this relation, provided that the corresponding invariant holds in the pre-state:

Lemma 5 For every act , if $\mathcal{I}_{act}(n, R, W)$, $n \xrightarrow{l} n'$ and $l \in L(act)$ then $n \sqsubseteq_{act} n'$.

Proof This is immediate for initialization since the initialization automaton does not modify $n.reg.BD_RAM$.

For transmission and reception, the first conjunct of $n \sqsubseteq_{act} n'$ holds since the corresponding automaton does not modify the NDP fields of the BDs in $q_{act}^{nic}(n)$, and $q_{act}^{nic}(n)$ contains no overlapping BDs (by $\mathcal{I}_{act}(n, R, W)$). The second conjunct holds since the automaton assigns only fields of BDs in $q_{act}^{nic}(n)$ (by $\mathcal{I}_{act}(n, R, W)$). $\square$

The next Lemma states that each action either shrinks the corresponding queue or does not modify its location:

Lemma 6 For every act , if $\mathcal{I}_{act}(n, R, W)$, $n \xrightarrow{l} n'$, and $l \in L(act)$ then $\exists q. q_{act}^{nic}(n) = q \cdot q_{act}^{nic}(n')$.

Proof $\mathcal{I}_{act}(n, R, W)$ implies that $q_{act}^{nic}(n)$ contains no overlapping BDs. In addition, no automaton assigns an NDP field of a BD. Therefore, no automaton can change the location of the BDs in its queue. If the state component identifying the head of $q_{act}^{nic}(n.tx.start-bda)$ in the case $act = tx$ is not modified, the location of the queue is not modified; and if that state component is modified, then it is set to either 0 (emptying the queue), or to the next BD which is a member of $q_{act}^{nic}(n)$ (by $\mathcal{I}_{act}(n, R, W)$; shrinking the queue). $\square$

We finally show that each action preserves the invariant of the other actions via a corollary:

Corollary 1 For every $act \neq act'$, if $\mathcal{I}_{NIC}(n, R, W)$, $n \xrightarrow{l} n'$, and $l \in L(act)$ then $n \succeq_{act'} n'$.

Proof Assume $act \in \{tx, rx\}$ and $act' = it$. Lemma 1 gives $n \succeq_{act'} n'$, and Lemma 2 gives $n.atm.s = idle \implies n'.atm.s = idle$ for $atm \in \{tx, rx\}$. Therefore, $n \succeq_{it} n'$ holds.

If $act = it$ then the transition is performed by the initialization automaton, which does not modify $n.tx$, $n.rx$ (by Lemma 1), nor $n.reg.BD_RAM$. Therefore q_{tx}^{nic} and q_{rx}^{nic} are unchanged.

If $act = tx$ and $act' = rx$, then Lemma 1, Lemma 5, and Lemma 6 imply $n \succeq_{rx} n'$. The same reasoning applies for $act = rx$ and $act' = tx$. $\square$

Corollary 2 For every $act \neq act'$, if $\mathcal{I}_{NIC}(n, R, W)$, $n \xrightarrow{l} n'$, and $l \in L(act)$ then $\mathcal{I}_{act'}(n', R, W)$.

Proof Follows from Corollary 1 and Lemma 4. $\square$

Corollary 3 For every act , if $\mathcal{I}_{NIC}(n, R, W)$, $n \xrightarrow{l} n'$, and $l \in L(act)$ then $\mathcal{I}_{qs}(n', R, W)$.

Proof Lemma 5, Lemma 1 and $\mathcal{I}_{qs}(n, R, W)$ imply that an action cannot modify the queue of another action. This property, Lemma 6, and $\mathcal{I}_{qs}(n, R, W)$, imply that the queues remain disjoint. $\square$

7 HOL4 implementation

Verifying correctness of the invariant requires handling a large state space, since it depends on the actual binary content of the BDs. This prevents the usage of model checkers, because they cannot enumerate all possible values of the BD_RAM . For this reason, the model and the proof have been implemented using the HOL4 interactive theorem prover [16]. Hereafter we briefly summarize some details of the implementation.

The HOL4 model uses an oracle to decide which automaton shall perform the next NIC transition and to identify properties of received frames (e.g., when a frame is received, its content, and presence of CRC errors). The oracle is also used to resolve some of the ambiguities in the NIC specification [1].

The NIC transition relation is defined in terms of several functions, one for each automaton state. In HOL4 $n \xrightarrow{l} n'$ is represented as $n' = \delta_{n.atm.s}^{atm}(n)$, where atm is the automaton performing the transition l and $\delta_{n.atm.s}^{atm}$ is the transition function of atm from the state $n.atm.s$.

The implementation of the proof of Lemma 5 is based on the following strategy:

1. For each BD field f we introduce a HOL4 function, $w_i(BD_RAM, a, v)$, which writes the value v to the BD

field f in BD_RAM of the BD at address a , and returning the resulting representation of BD_RAM .

- The HOL4 function *write* performs several BD field writes sequentially:

$$\begin{aligned} \text{write}([], BD_RAM) &:= BD_RAM \\ \text{write}([(w_1, a_1, v_1), \dots, (w_k, a_k, v_k)], BD_RAM) \\ &\quad \text{write}([(w_2, a_2, v_2), \dots, (w_k, a_k, v_k)], \\ &:= w_1(BD_RAM, a_1, v_1)) \end{aligned}$$

- For each transition function δ_s^{atm} , we define a (possibly empty) list $W_s^{atm}(n) = [t_1(n), \dots, t_k(n)]$, whose elements $t_i(n)$ are triples of the form (w, a, v) , depending on the state n , and in which w , a and v denote, respectively, a function writing a BD field, an address, and a value. We prove that δ_s^{atm} and W_s^{atm} update $n.reg.BD_RAM$ identically:

$$\begin{aligned} \delta_s^{atm}(n).reg.BD_RAM \\ = \text{write}(W_s^{atm}(n), n.reg.BD_RAM) \end{aligned}$$

For tx and rx , we also prove that the written BDs are in the corresponding queue ($\{t_1.a, \dots, t_k.a\} \subseteq q_{atm}(n)$), and for td and rd that the written BD is the BD following the last processed BD ($\{t_1.a, \dots, t_k.a\} \subseteq \{n.tx.bda\}$ and $\{t_1.a, \dots, t_k.a\} \subseteq \{n.rx.bda\}$ respectively).

- We prove that each w_i writes only the BD at the given address a and preserves the NDP field:

$$\begin{aligned} (\forall a' \notin \mathcal{A}([a]). \\ BD_RAM(a') = w_i(BD_RAM, a, v)(a')) \wedge \\ bd(BD_RAM, a).ndp \\ = bd(w_i(BD_RAM, a, v), a).ndp \end{aligned}$$

- Finally, we prove Lemma 5 for every update $\text{write}(W_s^{atm}(n), n.reg.BD_RAM)$, provided that all possible pairs of BDs at the addresses in $W_s^{atm}(n)$ are non-overlapping (that is, the BDs at locations $t_i.a$ and $t_j.a$ do not overlap for $\{t_i, t_j\} \subseteq W_s^{atm}(n)$). The non-overlapping is implied by $\mathcal{I}_{NIC}(n, R, W)$.

HOL4 requires a termination proof for every function definition. For this reason the function $q(n, a)$ (i.e., the list of addresses of reachable BDs from address a in the NIC state n) cannot be implemented by recursively traversing the BDs by reading their NDP. In general the linked list can be cyclic and therefore the queue can be infinite. This problem is solved as follows. We introduce a predicate $BD_Q(q, a, BD_RAM)$ that holds if the queue q is the list (which is finite by definition in HOL4) of addresses of BDs in BD_RAM starting at address a , linked via the NDP fields, and containing a BD with an NDP field equal to 0 (the last BD). This predicate is defined by structural induction on

the list q and its termination proof is therefore trivial. We show that the queue starting from a given address in a given BD_RAM is unique:

$$\begin{aligned} \forall q \, q' \, a \, BD_RAM. \\ BD_Q(q, a, BD_RAM) \wedge BD_Q(q', a, BD_RAM) \\ \implies q' = q \end{aligned}$$

$\mathcal{I}_{tx-wd}$ includes a conjunct stating that the transmission queue is not circular. That conjunct is phrased in HOL4 as there exists a list q satisfying $BD_Q(q, n.tx.start-bda, n.reg.BD_RAM)$. This enables a definition of q_{tx}^{nic} by means of Hilbert's choice operator applied on the set

$$\{q \mid BD_Q(q, n.tx.start-bda, n.reg.BD_RAM)\}$$

(the choice operator returns an arbitrary element of the set satisfying the predicate). Since this set contains only one element, a unique queue is returned satisfying the predicate. The same approach is used for the reception queue.

The model of the NIC consists of 1500 lines of HOL4 code. Understanding the NIC specification, experimenting with hardware, and implementing the model required (roughly) three man-months of work. The NIC invariant consists of 650 lines of HOL4 code and the proof consists of approximately 55000 lines of HOL4 code (including comments). Identifying the invariant, formalizing it in HOL4, defining a suitable proof strategy, and implementing the proof in HOL4 required (roughly) one man-year of work. Executing the proof scripts take approximately 45 minutes on a 2.93GHz Xeon(R) CPU X3470 with 16GB RAM.

8 Isolating secure partitions in an IoT system

To demonstrate the applicability of our design we developed a software platform to isolate security critical components from a connected Linux system. BeagleBone Black is used for evaluation.

8.1 Existing platform

Prosper (c.f. Fig. 6a) is a hypervisor [12] for ARMv7 that is capable of isolating a Linux guest from itself and other guests. The latter can be used to deploy security critical software and isolate it from faults in Linux. Linux is paravirtualized (modified) to be executed in user mode alongside its applications. Only the hypervisor is executed in privileged mode and which is invoked via hypercalls. In order to guarantee isolation, the hypervisor is in control of the MMU and virtualizes the memory subsystem via direct paging: Linux allocates the page tables inside its own memory area and can

Fig. 6 Prosper hypervisor

directly modify them while the tables are not in active use by the MMU; once the page tables are in active use by the MMU, the hypervisor guarantees that those page tables can be modified only via hypercalls. The isolation properties of Prosper have been formally verified. However, the existing proofs disregard devices and assume that memory can be changed only by the CPU whose accesses are mediated by the MMU.

8.2 Attacker model

Concerning the Linux guest it is not realistic to restrict the attacker model, since it has been repeatedly demonstrated that software vulnerabilities have enabled overtaking complete Linux systems via privilege escalation. For this reason we assume that the attacker has complete control of the Linux guest. The attacker can force Linux to execute and access arbitrary code and data. It is assumed that the goal of the attacker is to escape isolation, i.e., reading or writing arbitrary memory of a secure guest.

Prosper guarantees isolation (i.e., prevents direct information flow between Linux and a secure guest) if the CPU is the only hardware component that can access memory [5]. However, if Linux can configure a DMA device then Linux can indirectly perform arbitrary memory accesses with catastrophic consequences: for example it can configure BDs to address hypervisor code, page tables, secure guest memory, and confidential memory regions, all of which will be written or read by the DMA device.

8.3 Secure network connectivity via monitoring

We extend the system with Internet connectivity while preventing Linux from abusing the DMAC of the NIC. We deploy a NIC monitor (c.f. Sect. 9) within the hypervisor that validates all NIC reconfigurations (c.f. Fig. 6b). The hypervisor forces Linux to map the NIC registers with read-only access (NIC register reads have no side effects). When the Linux NIC driver attempts to configure the NIC, by

Table 1 Netperf benchmarks. TS (TCP_STREAM), TM (TCP_MAERTS) and US (UDP_STREAM) are measured in Mbit/second, and TR (TCP_RR) and UR (UDP_RR) are measured in transactions/second

Configuration	Benchmark				
	TS	TM	US	TR	UR
Native	94.1	93.9	96.2	3365.1	3403.4
Native + monitor	94.1	93.9	96.2	3317.6	3402.2
Hyper	16.2	45.6	29.3	924.9	1009.0
Hyper + monitor	15.3	41.0	27.6	891.3	982.6

writing a NIC register, an exception is raised. The hypervisor catches the exception and, in case of a NIC register write attempt, invokes the monitor. The monitor checks whether the write preserves the NIC invariant, and if so re-executes the write, and otherwise blocks it. In addition to the NIC monitor, we extended the checks of the hypervisor to ensure that page tables are not allocated in buffers address by BDs in the reception queue, since those buffers are written when frames are received.

Having the NIC driver in Linux in contrast to a specialized NIC driver in the hypervisor has several advantages. It keeps the code of the hypervisor small, and avoids verification of code that manages power management, routing tables and statistics of the NIC. Furthermore, in this design the interface between the OS and the NIC is OS independent. The monitor provides a NIC interface that closely mimics that of the NIC, with the difference that security violating reconfigurations are blocked. Hence, the hypervisor and the monitor can be used with different OSs, OS versions, and device driver versions. Finally, the design demonstrates a general approach to secure DMACs that are configured via linked lists of BDs and can easily be adapted to support other DMACs.

8.4 Evaluation

We evaluated network performance with netperf for the system in Fig. 6b, involving Linux 3.10 and BeagleBone Black (BBB). Linux was running netperf 2.7.0 on BBB, which was connected with a 100 Mbit Ethernet point-to-point link to a PC running netperf 2.6.0. The benchmarks are: TCP_STREAM and TCP_MAERTS transfer data with TCP from BBB to the PC and vice versa; UDP_STREAM transfers data with UDP from BBB to the PC; and TCP_RR and UDP_RR use TCP and UDP, respectively, to send requests from BBB and replies from the PC. Each benchmark lasted for ten seconds and was performed five times. Table 1 lists the average value for each test.

We compare the network performance of the system (hyper + monitor) shown in Fig. 6b with the system

(hyper) where Linux is executed on top of the prosper hypervisor but is free to directly configure the NIC, and therefore being able to violate all security properties. The performance of the hyper + monitor system is between 89.9% and 97.4% of the Hyper system. This performance loss is expected due to the additional context switches caused by the Linux NIC driver attempting to write NIC registers.

To validate the monitor design we also experimented with a different system. In this case we consider a trusted Linux kernel that is executed without the hypervisor but with a potentially compromised NIC driver (Native). This is typically the case when the driver is a binary blob. In order to prevent the driver from abusing the NIC DMA the monitor is added to the Linux kernel (native + monitor). The Linux NIC driver has been modified to not directly write NIC registers but instead to invoke the monitor when it needs to write a NIC register. The monitor is similar to the one in the hypervisor, and the C file containing the monitor code is located in the same directory as the Linux NIC driver. The overhead introduced by this configuration is negligible, as demonstrated by the first two lines of Table 1. The same approach can for instance be used to monitor an untrusted device driver that is executed in user mode on top of a microkernel (e.g., seL4 and Minix).

In addition to being OS and NIC driver independent, the monitor minimizes the trusted computing base configuring the NIC. In fact, the monitor consists of 900 lines of C code while the Linux NIC driver consists of 4650 lines. Moreover, the monitor is independent of the specific version of the Linux kernel and the NIC driver, the latter having grown to 6500 lines in Linux 5.2.

9 NIC monitor

This section describes the NIC monitor of Fig. 6b. The top-level function *check_write*(*v*, *pa*) of the monitor is invoked when Linux attempts to write the 4-byte word value *v* to the NIC register at the physical address *pa*. Physical addresses are used instead of virtual addresses to make the monitor independent of the virtual address map. First, *check_write* checks that the address is 4-byte aligned, to ensure that exactly one register is accessed. Then, *check_write* checks which NIC register is located at *pa* and invokes the corresponding handler. Each handler performs the write if the write preserves the NIC invariant $\mathcal{I}_{NIC}$. Each handler returns true only if the write preserves the NIC invariant. The returned truth value is used by the hypervisor to take a suitable action in case Linux does something suspicious.

The monitor uses the following data structures to track the state of the NIC:

- *init* is a boolean variable indicating that the NIC is initialized.
- *cleared*[*p*] is an array of booleans indicating if the register *p* has been cleared during the initialization procedure, where *p* ranges over the four transmission and reception HDP and CP registers.
- *tx_td*, *rx_td* are booleans indicating if the NIC is performing a teardown operation.
- *tx_s*, *rx_s* are pointers to the head of the NIC queues.
- *active_bd*[*a*] is a mask indicating if the word of BD_RAM at address *a* stores is part of a BD that reachable from *tx_s* or *rx_s*. This masks is used to optimize checks of writes to BD_RAM.

The following describes the the support function *update_q* and the handlers of the monitor.

9.1 Subroutine *update_q*

The datastructures *tx_s*, *rx_s* and *active_bd* must be periodically updated by the monitor to “release” the BDs that have been processed by the NIC. This “garbage collection” is performed by the subroutine *update_q*. The argument of the subroutine can be *tx* or *rx* to indicate with queue must be analyzed. We describe the behavior for *tx*, since the case for *rx* is analogous. If the register TX_CP is 0xFFFFFCC, then the NIC has finished transmission teardown, meaning that transmission is idle and the corresponding queue is empty. In this case, *update_q* traverses the BDs starting at *tx_s/rx_s*, unmarks each corresponding entries in *active_bd*, and sets *tx_s* to 0. Otherwise, this traversal is done up to the first BD whose OWN flag is set, and *tx_s* is set to the address of that BD.

Figure 7 illustrates this process. Each state is represented by two columns, one column for the NIC state and one column for the monitor’s state. In the first state (columns 1 and 2), the NIC has transmission and reception queues, whose start addresses are identified by the internal NIC variables *tx_p* and *rx_p* (denoted in the NIC model by *n.tx.start-bda* and *n.rx.start-bda*). The start locations of the queues are recorded by the monitor variables *tx_s* and *rx_s* and the addresses used by these queues marked in *active_bd*. The second state (columns 3 and 4) shows the result of the NIC transmitting the first two BDs. The internal NIC variable *tx_p* is advanced to address the third BD in the transmission queue. The monitor’s variable *tx_s* is now lagging behind the transmission queue and *active_bd* marks some BDs that have been already processed. However, *tx_s* still identifies a queue of which the transmission queue is a suffix. The execution of *update_q* collects these BDs, by updating *tx_s* to point to the current head of the transmission queue and unmaking from *active_bd* the traversed BDs.

Fig. 7 BDs in BD_RAM are marked in grey. Each black square of *active_bd* represents a location that the monitor considers in use for BDs

9.2 Handler reset (Fig. 8)

This handler is normally invoked when Linux attempts to trigger the reset operation of the NIC by writing 1 to RESET.

If the value to write to RESET is 0, then the monitor accepts the request because this operation has no effect. Otherwise, the monitor checks whether the NIC is being currently initialized ($\neg init$), is tearing down an operation (tx_td or rx_td). If so, the request is rejected, since the effect of writing RESET while the NIC is performing any of these operations is unspecified. If the checks succeed, 1 is written to RESET to start the reset operation. In addition, the data structures tracking the initialization procedure of the NIC are set to false.

9.3 Handlers *tx_hdp* (Fig. 9) and *rx_hdp*

The handler *tx_hdp* is invoked either during initialization (to clear TX_HDP by writing 0) or to start transmission (by writing the address of the first BD of the new queue).

Clearing TX_HDP is allowed only if the NIC is currently being initialized ($\neg init$), the internal reset operation has been completed (RESET = 0), and the attempted write

```

function bool : reset(v : word)
  if v = 0 then
    return true
  else if  $\neg init \vee tx\_td \vee rx\_td$  then
    return false
  else
    for p ∈ {tx_hdp, rx_hdp, tx_cp, rx_cp} do
      cleared[p] := false
    init := false
    RESET := 1
    return true

```

Fig. 8 Pseudocode of the handler for writes to RESET

```

function bool : tx_hdp(v : word)
  if  $\neg init$  then
    if RESET = 1  $\vee$  v  $\neq$  0 then
      return false
    else
      TX_HDP := 0
      cleared[tx_hdp] := true
      update_q(tx)
      if  $\bigwedge_p$  cleared[p] then
        init := true
        return true
  else
    if TX_HDP  $\neq$  0  $\vee$  tx_td then
      return false
    else
      update_q(tx)
      if is_q_secure(v, tx) then
        prepare_queue(v, tx)
        add_active_q(v, tx)
        TX_HDP := v
        return true
      else
        return false

```

Fig. 9 Pseudocode of the handler for writes to TX_HDP

clears the register ($v = 0$). If these conditions are satisfied, it is recorded that TX_HDP has been initialized. If all HDP and CP registers have been cleared then the initialization is complete. Therefore *initialization_performed* sets *init* to true, and clears *tx_s*, *rx_s* and *active_bd* to records that there is no BD in use by the NIC.

Starting transmission is allowed only if the NIC has been initialized (*init*), transmission teardown is not being performed ($\neg tx_td$), and TX_HDP is 0. This means that the NIC is not transmitting and the transmission queue is empty. If these conditions are satisfied, *update_q* is invoked to garbage collect old transmission BDs. Notice that *update_q* sets *tx_s* to 0 since TX_HDP is 0. Then *is_q_secure* checks that the transmission queue starting at *v* is secure. Namely the following conditions must be satisfied:

1. BDs are located at 4-byte aligned addresses in BD_RAM, and do not overlap the transmission or reception queues (the former is empty since TX_HDP = 0). The latter condition is checked by a lookup in *active_bd*.
2. No pairs of BDs overlap.
3. Each BD is well-formed (e.g., each BD is both SOP and EOP, and the OWN flag is cleared).
4. BDs address only readable memory.

If these conditions are satisfied, *prepare_queue* sets the OWN flag and clears the EOQ flag of each BD of the new queue. Then *add_active_q* sets *tx_s* to the address of the first BD of the new queue (*v*) and marks the entries of the new

BDs in *active_bd*. Finally, the monitor starts transmission by writing the address of the first BD to TX_HDP.

9.4 Handler *bd_ram* (Fig. 10)

The handler *bd_ram* is normally invoked in two situations. The first case is when Linux is initializing some fields of a new BD that will be later given to the NIC (either as an extension of an existing queue, or as a new queue by writing TX_HDP or RX_HDP). The second situation is when Linux is attempting to extend a queue by writing the NDP field of the last BD of the queue. The handler always uses *update_q(tx)* and *update_q(rx)* to garbage collect transmission and reception BDs.

In the first situation, the address of the BD to initialize cannot be already in use by the NIC, hence *active_bd[pa]* must be unmarked. In this case, the monitor updates BD_RAM, by writing *v* in the address *pa* (*address_space[pa] := v* in Fig. 10).

In the second situation (*active_bd[pa]*) the attempted write targets a BD in use by the NIC. Function *q_access* traverses the queues starting at *tx_s* and *rx_s* to check if the attempted write addresses the NDP field of the last BD of the transmission or reception queue (*q_access(pa) ∈ {tx, rx}*). In this case, the monitor performs the same operations as in the handlers *tx_hdp* and *rx_hdp* (depending on whether the transmission or reception queue is to be extended), with the exception that BD_RAM is written instead of TX_HDP and RX_HDP. If the attempted write targets any other part of the queues (i.e., the NDP field of the corresponding BD is not 0 or *pa* points to other fields of an existing BD), then Linux is attempting to modify a BD that is currently in use by the

```

function bool : bd_ram(pa : word, v : word)
  update_q(tx)
  update_q(rx)

  if ¬active_bd[pa] then
    address_space[pa] := v
    return true

  q_access_type := q_access(pa)
  if q_access_type ∈ {tx, rx} then
    if is_q_secure(v, q_access_type) then
      prepare_queue(v, q_access_type)
      add_active_q(v, q_access_type)
      address_space[pa] := v
      return true
    else
      return false
  else
    return false

```

Fig. 10 Pseudocode of the handler for writes to BD_RAM

NIC. This operation is forbidden by the monitor, irrespectively of whether it preserves the security conditions.

9.5 Handlers *tx_cp* (Fig. 11) and *rx_cp*

The handler is invoked in two situations. In the first case Linux is attempting to clear and initialize TX_CP and the monitor behave analogously to the case of clearing TX_HDP for *tx_hdp*. In the second case, the handler is only used to detect the completion of a transmission teardown. The monitor releases the transmitted BDs and updates *tx_td* if transmission teardown has been completed.

The handler *rx_cp* operates in the same way, but with respect to reception instead of transmission.

9.6 Handlers *tx_td* (Fig. 12) and *rx_td*

This handler is invoked when Linux attempts to teardown transmission by writing 0 to TX_TD. It is unspecified to initiate a transmission teardown operation while the NIC is currently being initialized (*init*) or is already performing a transmission teardown operation (*tx_td*). If the teardown request is accepted, 0 is written to TX_TD to activate a teardown and *tx_td* is set to *true*. The handler *rx_td* is identical but handles to reception instead of transmission.

```

function bool : tx_cp(v : word)
  if ¬init then
    if RESET = 1 ∨ v ≠ 0 then
      return false
    else
      TX_CP := 0
      cleared[tx_cp] := true
      if ∧p cleared[p] then
        init := true
        return true
      return true
    else
      update_q(tx)
      if TX_CP = 0xFFFFFFFF then
        tx_td := false
        return true
      else
        return false

```

Fig. 11 Pseudocode of the handler for writes to TX_CP

```

function bool : tx_td(v : word)
  if ¬init ∨ tx_td then
    return false
  tx_td := true
  TX_TD := 0
  return true

```

Fig. 12 Pseudocode of the handler for writes to TX_TD

9.7 Default handler

The default handler simply prevents writes to all other NIC registers, which are not used by the Linux NIC driver.

10 Correctness of the NIC monitor

This section presents a semi-formal analysis of the correctness of the monitor. Independently of the argument values v and pa given to $check_write(v, pa)$, $check_write$ should preserve $\mathcal{I}_{NIC}$. We analyze each handler individually. Since $\mathcal{I}_{NIC}$ only depends on the NIC state, the monitor can only violate $\mathcal{I}_{NIC}$ by writing the NIC registers (c.f. rule in Sect. 4 involving NIC transitions with the label *update*).

10.1 Monitor invariant

Clearly correctness of the monitor depends on its data structures correctly tracking the state of the NIC. This property is formulated as an invariant $\mathcal{I}_{MON}(m, n)$, which relates the state (data structures) of the monitor m with the state of the NIC. The most important parts of $\mathcal{I}_{MON}$ are:

- $m.init \iff n.it.s = idle$: $init = true$ if and only if the NIC is initialized.
- $cleared[p] = true$ if and only if the corresponding HDP/CP register p has been initialized/cleared during the current initialization.
- $n.tx.s \neq idle \implies m.tx_td$: If the NIC is performing a transmission teardown then $tx_td = true$. A corresponding invariant holds for reception teardown.
- $\exists q'. q' \cdot q_{tx}^{nic}(n) = q_{tx}^{mon}(m, n)$: The transmission queue of the NIC $q_{tx}^{nic}(n)$ is a suffix of the transmission queue of the monitor $q_{tx}^{mon}(m, n)$, where $q_{tx}^{mon}(m, n) := q(n, m.tx_s)$. A corresponding invariant holds for the reception queue.
- $\forall a \in \mathcal{A}(q_{tx}^{mon}(m, n)) \cup \mathcal{A}(q_{rx}^{mon}(m, n)).$
 $m.active_bd[word(a)]$:
 $active_bd$ marks which words of BD_RAM that store BDs reachable from tx_s or rx_s . $\mathcal{A}(bds)$ denotes the set of byte addresses of the bytes of the BDs in the queue bds , and $word(a)$ denotes the word-aligned address of the word containing the byte located at address a .

In the following we do consider each handler to be executed atomically. A formal proof for the monitor would require to show that transitions of the NIC interleaved with the monitor operation can be reordered without affecting the preservation of the invariant.

10.2 Subroutine *update_q* preserves $\mathcal{I}_{NIC} \wedge \mathcal{I}_{MON}$

The subroutine *update_q* does not write any NIC register, hence $n = n'$. The subroutine only affects tx_s (if argument is tx), rx_s (if argument is rx) and $active_bd$. As usual we analyze the case for transmission, since the reception case is similar. There are two possible scenarios depending on whether *update_q* reads $0xFFFFFFF$ from TX_CP .

Case 1 If $TX_CP = 0xFFFFFFF$ then all BDs in $q_{tx}^{mon}(m, n)$ are unmarked and tx_s is set to 0, implying $q_{tx}^{mon}(m', n) = []$. This case can only happen after that transmission teardown automaton has performed the transition from *write-cp* to *idle* which means $n.tx.start-bda = 0$, hence $q_{tx}^{nic}(n) = []$.

Case 2 If $TX_CP \neq 0xFFFFFFF$ then *update_q* sets tx_s to the address of the first BD reachable from $m.tx_s$ and whose OWN flag is set. $\mathcal{I}_{tx-wd}$ states that all BDs in $q_{tx}^{nic}(n)$ have their OWN flag set. Since *update_q* advances $m'.tx_s$ to the first BD in the queue with the OWN flag set, $q_{tx}^{nic}(n')$ remains a suffix of $q_{tx}^{mon}(m', n)$. Also, by the separation of the transmission and reception queue, we can infer that the traversed BDs are not part of the reception queue. Therefore they can be safely be unmarked in *active_bd*.

10.3 Handler *reset* preserves $\mathcal{I}_{NIC} \wedge \mathcal{I}_{MON}$

Note that this function affects the data structures of the monitor or the NIC state only if $v \neq 0$, $m.initialized$, $\neg tx_td$, and $\neg rx_td$. In this case, $\mathcal{I}_{MON}(m, n)$ ensures that the initialization and tear down automata are in the states *idle*. Writing 1 to RESET in this case causes the initialization automaton to enter the state *reset*. Since *init* and *cleared* are set to false (NIC nor any HDP or CP registers are initialized), $\mathcal{I}_{MON}$ is preserved. Regarding $\mathcal{I}_{NIC}$, only $\mathcal{I}_{wd}$ and $\mathcal{I}_{it}$ are relevant. $\mathcal{I}_{it}$ is preserved since $n'.it.s = reset \neq init_regs$. $\mathcal{I}_{wd}$ is preserved since writing RESET causes the NIC model to enter an undefined state only when the teardown or initialization automata are not idle.

10.4 Handler *tx_hdp* preserves $\mathcal{I}_{NIC} \wedge \mathcal{I}_{MON}$

There are two cases where *tx_hdp* affects the states of the monitor data structures or the NIC state:

1. $\neg init$, RESET = 0, and $v = 0$: TX_HDP is cleared during initialization.
2. $init$, TX_HDP = 0, $\neg tx_td$, and $is_q_secure(v, tx)$: TX_HDP is written with v to start transmission of a secure queue with the first BD at address v .

Case 1 $\neg m.init$, $n.reg.RESET = 0$ and $\mathcal{I}_{MON}(m, n)$ imply $n.it.s = init_regs$. Writing 0 to TX_HDP when the initialization automaton is in this state means that TX_HDP

has been initialized, and therefore $cleared[tx_hdp]$ is set to $true$. If all other HDP and CP registers have also been initialized (i.e., $\bigwedge_{p \neq tx_hdp} cleared[p]$ as implied by $\mathcal{I}_{MON}(m, n)$) then initialization is complete. Hence, $n'.its = idle$, and $init$ is therefore set to $true$. $\mathcal{I}_{MON}$ is therefore preserved.

Regarding $\mathcal{I}_{NIC}$, $\mathcal{I}_{wd}$ is preserved because clearing TX_HDP when the initialization automaton is in $init_regs$ do not cause the NIC to enter an undefined state. All other sub-invariants of $\mathcal{I}_{tx}$ hold vacuously since $q_{tx}^{nic}(n')$ is empty (since clearing TX_HDP causes the NIC model to also clear $n'.start_bda$).

Case 2 $TX_HDP = 0$ implies that $q_{tx}^{nic}(n)$ is empty, and $is_q_secure(v, tx)$ implies that the new transmission queue $q_{tx}^{nic}(n')$ starting at v , is not overlapping with the reception queue $q_{rx}^{nic}(n)$. For this reason, the writes of the OWN and EOQ fields by $prepare_queue(v, tx)$ does not affect the sub-invariants of $\mathcal{I}_{NIC}$ that depend on the reception queue. $is_q_secure(v, tx)$ implies that $q_{tx}^{nic}(n')$ satisfies all security requirements, and thus also all sub-invariants of $\mathcal{I}_{NIC}$ that depend on the transmission queue. Regarding $\mathcal{I}_{MON}$, $add_active_q(v, tx)$ marks all entries of $active_bd$ of the BDs in $q_{tx}^{nic}(n')$ and sets tx_s to v , thereby preserving their associated invariants, and thus also $\mathcal{I}_{MON}$.

10.5 Handler bd_ram preserves $\mathcal{I}_{NIC} \wedge \mathcal{I}_{MON}$

There are two cases in which the execution of bd_ram affects the data structures of the monitor or the NIC state:

1. pa does not address a BD reachable from tx_s or tx_s ($\neg active_bd[pa]$).
2. pa addresses an NDP field equal to zero of a BD in the transmission or reception queue.

Case 1 $\mathcal{I}_{MON}(m, n)$ implies that the transmission and reception queues of the NIC ($q_{tx}^{nic}(n)$ and $q_{rx}^{nic}(n)$) are suffixes of the corresponding queues as viewed by the monitor ($q_{tx}^{mon}(m, n)$ and $q_{rx}^{mon}(m, n)$). Since pa does not address a 4-byte word of a BD reachable from tx_s or tx_s , the addressed location is not a part of a BD in use by the NIC. The write does therefore not affect the NIC queues nor the NIC automata. That is, the write satisfies $n \geq_{act} n'$, keeps the queues disjoint and does not cause the NIC to enter an undefined state, thereby preserving each sub-invariant of $\mathcal{I}_{NIC}$ by Lemma 4. In addition, no data structure of the monitor is written, thereby preserving $\mathcal{I}_{MON}$.

Case 2 Only the case for transmission is considered since the case for reception is similar, for which the reasoning is nearly identical to Case 2 of the handler tx_hdp . The difference is that there is an existing transmission queue, which the appended queue is checked to not overlap.

10.6 Handler tx_cp preserves $\mathcal{I}_{NIC} \wedge \mathcal{I}_{MON}$

The operations of tx_cp and tx_hdp when $init = true$ are analogous and the correctness reasoning of tx_cp is therefore analogous to the correctness reasoning of tx_hdp .

Otherwise ($\neg init \wedge tx_td \wedge TX_CP = 0xFFFFFFFFC$), for $\mathcal{I}_{MON}$ to be preserved, the transmission teardown automaton must be in the state $idle$. The only transition of the NIC model that writes $0xFFFFFFFFC$ to TX_CP is the last transition of the transmission teardown operation. Hence, if $TX_CP = 0xFFFFFFFFC$, the transmission teardown automaton is idle.

10.7 Handler tx_td preserves $\mathcal{I}_{NIC} \wedge \mathcal{I}_{MON}$

Writing TX_TD has two possible outcomes depending on the state of the NIC and the value written. If either the NIC is not initialized ($n.its \neq init_regs$), the transmission teardown automaton is not idle ($n.td.s \neq idle$), or the value written is not 0, then the NIC model enters an undefined state. Otherwise, the transmission teardown automaton is activated.

The former outcome cannot occur since only 0 is written to TX_TD and that write only occurs when $init$ and $\neg tx_td$. The values of the latter two monitor data structures and $\mathcal{I}_{MON}(m, n)$ imply that the NIC is initialized and that the transmission teardown automaton is in the state $idle$. Therefore the NIC does not enter an undefined state. Hence, only the latter outcome is relevant, causing activation of the transmission teardown automaton, which does not affect $\mathcal{I}_{NIC}$ and thus $\mathcal{I}_{NIC}$ is preserved. Since tx_td is set to $true$, $\mathcal{I}_{MON}$ is preserved.

11 Application: prevention of code injection and secure system upgrade

We demonstrate the platform of Sect. 8 by extending the functionalities of an existing application. MProsper [5] uses the Prosper hypervisor to prevent code injection in the untrusted Linux (c.f. Fig. 13a). Mprosper uses the isolated partition to execute a Virtual Machine Introspector (VMI) and code hashing. This partition prevents execution of code (i.e., memory page) whose hash value is not in the database of trusted program hashes, referred to as the “golden image”.

The hypervisor supervises all modifications of the page tables and informs MProsper of all modifications of the virtual memory layout. Whenever Linux (1) requests to change a page table, (2) the hypervisor identifies the physical pages that are requested to be made executable (if the request involves executable permissions) and requests their validation to MProsper. The VMI (3) computes the hash values of those pages, and checks that the hash values are in the

Fig. 13 Preventing code injection by means of MProspers. A dashed box represents that the memory region is read-only of the corresponding software component

golden image. The hypervisor (4) applies the changes only if the checks of MProspers succeed. Additionally, MProspers forces Linux to obey the executable space protection policy: A memory page can be either executable or writable, but not both. These policies guarantee that the hash values of the code have been checked by MProspers before the code is executed and that executable code remains unmodified after validation.

In the considered scenario, the attacker has the goal of executing arbitrary binary programs via any vulnerability of the compromised Linux guest. Similarly to the hypervisor, MProspers prevents these attacks if the CPU is the only hardware component that can modify memory [5]. If Linux can configure DMA accesses, the compromised Linux can modify the golden image or inject code into its own executable memory.

We modified MProspers to use the design of Fig. 13b. We extended the checks of MProspers and the NIC monitor to ensure that executable code is not allocated in buffers addressed by BDs in the reception queue (i.e., executable code is not located in *W*). This prevents a compromised Linux from exploiting the DMA accesses to bypass the code signature checks while enabling Internet connectivity to Linux applications.

This system design also enables connectivity to the secure components, which can use Linux as an untrusted “virtual” gateway. We used this feature to implement secure remote upgrade of Linux applications (c.f. Fig. 14). First, the hash values of the new binary code are computed and signed using the administration private key and then published by a remote host. Linux (1–2) downloads the new code, hash values and the associated signature and (3) requests an update of the golden image via a hypercall. The hypervisor forwards the request to MProspers. The signature (4) is checked by MProspers using the administration public key, and if it is valid, the golden image is updated with the new hash values. The use of digital signatures makes the upgrade trustworthy, even though Linux acts as a network intermediary,

Fig. 14 Secure remote upgrade

and furthermore, even if Linux is compromised. A similar approach is used to revoke hash values from the golden image.

12 Related work

Several projects have done pervasive verification of low level execution platforms (e.g., [6, 9–11, 19]). These projects usually do not take I/O devices into account. If I/O devices are taken into account then there are four approaches to show security properties of these platforms: (1) block disallowed memory accesses by disabling DMA or using explicit hardware support, like IOMMU for x86 (e.g., Vasudevan et al. [18]); (2) verify a privileged device driver; (3) monitor the configurations established by an untrusted and unprivileged device driver; and (4) synthesize a driver that is correct by construction. In the last three cases formal models of the I/O devices (the NIC in our case) are necessary.

Alkassar et al. [2] and Duan [7] have verified device drivers for UART devices. Alkassar et al. [3] have verified a page fault handler of a microkernel that controls an ATA disk, proving that after the driver has terminated, a specific page in memory has been copied to a sector of the disk. In all these cases, data transfers to and from the device occur via the CPU and no DMA is involved, therefore these devices do not constitute a threat to memory isolation.

The system design presented in [20] is similar to the system design of Fig. 13.a and consists of a hypervisor, a monitor, and untrusted guests. The hypervisor is based on XMHF [18] and configures the hardware to protect: the hypervisor from the monitor and from the guests; the monitor from the guests; and the guests from each other. The monitor (called wimpy kernel) checks device configurations built by guests to ensure isolation. Although memory integrity of the hypervisor has been verified, I/O devices are not considered in the verification since their memory accesses are checked by an IOMMU.

Device driver synthesis is a method for automatically generating device drivers that are correct by construction.

Some of these methods (e.g., [13, 14]) require a specification of the protocol of the communication between the OS and the device driver and between the device driver and the I/O device. Current results cannot synthesize device drivers for I/O devices with DMA. When only security properties are needed (and not functional correctness), communication protocols are not necessary for synthesis and a security invariant can be used to drive the synthesis of a run-time monitor (e.g., generation of a debugging monitor [17]).

13 Concluding remarks

We modeled the NIC of an embedded system and demonstrated that the NIC can be securely isolated. Isolation is formally verified by means of an invariant, which is preserved by all NIC operations, and which implies that all memory requests address only readable and writable memory regions. The invariant provides a blueprint for securing the NIC: Either the device driver ensures preservation of the invariant, or a run-time monitor is used to prevent potentially compromised software to violate the invariant. We demonstrated that the second method is practical, by developing and analyzing a run-time monitor and evaluating its deployment in a secure hypervisor.

The verification identified some properties of secure NIC configurations that are not explicitly stated by the specification and that may be overlooked by developers. For example, a queue must not contain overlapping BDs, since that could cause the NIC to modify the BP field of a BD when updating the OWN field of an overlapping BD.

We also identified a bug in the Linux driver while testing the monitor. When the driver module is unloaded, the driver (1) tears down reception; (2) frees the buffers in memory used for reception; (3) inadvertently re-enables reception; and (4) shuts down the DMA of the NIC. If a frame is received between (3) and (4), then the NIC writes into a freed buffer. In case of interrupt or parallel execution, this buffer may have been (re-)allocated to another software component, potentially causing data corruption. Moreover, this write after free can leak frame data to other software components. Finally, the thorough analysis of the NIC lead to the identification of self-contradictory statements in the NIC specification which have been reported in Sect. 5.

Our approach can be adapted to secure other DMACs that are configured via linked lists. In case the linked lists are stored in memory instead of being stored in the DMAC, then their elements must not overlap with writable buffers addressed by the BDs or be directly writable by untrusted software. Also, the linked lists cannot reside in non-readable memory, since the DMAC can then leak their configuration/content. In case the DMAC does not modify BDs, the constraint of non-overlapping BDs and queues is not needed.

Our approach can also handle register based DMACs by considering the registers used to configure memory accesses as a fixed queue. On the other hand, a general treatment of programmable DMACs is challenging, since they require a formal model of their instruction set, which can be used to define arbitrary behavior.

Acknowledgements Open access funding provided by Royal Institute of Technology. Work partially supported by the TrustFull project financed by the Swedish Foundation for Strategic Research.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. AM335x and AMIC110 Sitara Processors (2010) Technical Reference Manual, Texas Instruments Rev. P. <https://www.ti.com/lit/ug/spruh73p/spruh73p.pdf>
2. Alkassar E, Hillebrand M, Knapp S, Rusev R, Sergey T (2007) Formal device and programming model for a serial interface. In: Proceedings of VERIFY, pp 4–20
3. Alkassar E, Hillebrand MA (2008) Formal functional verification of device drivers. In: Conference on verified software: theories, tools, and experiments, Springer, pp 225–239
4. Cavada R, Cimatti A, Jochim CA, Keighren G, Olivetti E, Pistore M, Roveri M, Tchaltev A (2005) Nusmv 2.4 user manual: CMU and ITC-irst
5. Chfouka H, Nemati H, Guanciale R, Dam M, Ekdahl P (2015) Trustworthy prevention of code injection in linux on embedded devices. In: Esorics, Springer, pp 90–107
6. Dam M, Guanciale R, Khakpour N, Nemati H, Schwarz O (2013) Formal verification of information flow security for a simple ARM-based separation kernel. In: CCS, ACM, pp 223–234
7. Duan J (2013) Formal verification of device drivers in embedded systems. The University of Utah
8. Fox A, Myreen MO (2010) A trustworthy monadic formalization of the armv7 instruction set architecture. In: International conference on interactive theorem proving, Springer, pp 243–258
9. Gu R, Shao Z, Chen H, Wu XN, Kim JSV, Costanzo D (2016) CertiKOS: an extensible architecture for building certified concurrent OS kernels. In: OSDI, USENIX, pp 653–669
10. Klein G, Elphinstone K, Heiser G, Andronick J, Cock D, Derrin P, Elkaduwe D, Engelhardt K, Kolanski R, Norrish M et al (2009) seL4: formal verification of an OS kernel. In: Operating systems principles, ACM, pp 207–220
11. Leinenbach D, Santen T (2009) Verifying the microsoft hyper-V hypervisor with VCC. In: Proceedings of FM, pp 806–809
12. Nemati H, Guanciale R, Dam M (2015) Trustworthy virtualization of the armv7 memory subsystem. In: International conference on

- current trends in theory and practice of informatics, Springer, pp 578–589
13. Ryzhyk L, Chubb P, Kuz I, Le SE, Heiser G (2009) Automatic device driver synthesis with termite. In: SIGOPS, ACM, pp 73–86
 14. Ryzhyk L, Walker A, Keys J, Legg A, Raghunath A, Stumm M, Vij M (2014) User-guided device driver synthesis. In: OSDI, USENIX, pp 661–676
 15. Schwarz O, Dam M (2014) Formal verification of secure user mode device execution with dma. In: Haifa Verification Conference, Springer, pp 236–251
 16. Slind K, Norrish M (2008) A brief overview of HOL4. In: International conference on theorem proving in higher order logics, Springer, pp 28–32
 17. Sun J, Yuan W, Kallahalla M, Islam N (2005) Hail: a language for easy and correct device access. In: Proceedings of the 5th ACM international conference on embedded software, ACM, pp 1–9
 18. Vasudevan A, Chaki S, Jia L, McCune J, Newsome J, Datta A (2013) Design, implementation and verification of an extensible and modular hypervisor framework. In: SP, IEEE, pp 430–444
 19. Verbeek F, Havle O, Schmaltz J, Tverdyshev S, Blasum H, Langenstein B, Stephan W, Wolff B, Nemouchi Y (2015) Formal api specification of the pikeos separation kernel. In: NASA formal methods symposium, pp 375–389
 20. Zhou Z, Yu M, Gligor VD (2014) Dancing with giants: Wimpy kernels for on-demand isolated i/o. In: 2014 IEEE symposium on security and privacy, IEEE, pp 308–323