Formal Methods in Computer-Aided Design 2022

Formally Verified Isolation of DMA

Jonas Haglund
dept. TCS
KTH Royal Institute of Technology
Stockholm, Sweden
jhagl@kth.se

Roberto Guanciale
dept. TCS
KTH Royal Institute of Technology
Stockholm, Sweden
robertog@kth.se

***Abstract*—Every computer having a network, USB or disk controller has a Direct Memory Access Controller (DMAC) which is configured by a driver to transfer data between the device and main memory. The DMAC, if wrongly configured, can therefore potentially leak sensitive data and overwrite critical memory to overtake the system. Since DMAC drivers tend to be buggy (due to their complexity), these attacks are a serious threat.**

This paper presents a general formal framework for modeling DMACs and verifying under which conditions they are isolated. These conditions can be used as a specification for guaranteeing that a driver configures the DMAC correctly. The framework provides general isolation theorems that are common to all DMACs, leaving to the user only the task of verifying proof obligations that are DMAC specific. This provides a reusable verification infrastructure that reduces the verification effort of DMACs. Models and proofs have been developed in the HOL4 interactive theorem prover. To demonstrate the usefulness of the framework, we instantiate it with a DMAC of a USB.

***Index Terms*—formal verification, interactive theorem proving, DMA, I/O security, memory isolation**

I. Introduction

Direct memory access controllers (DMACs) are hardware components transferring data between memory and I/O devices (e.g. memory-to-memory copies, and data transfers to and from network interface cards, USB, disks, and graphics accelerators). Without a DMAC, the CPU must perform these data transfers, spending time on data transfers rather than on applications, decreasing performance significantly [1]–[3], [44]. DMACs can also reduce power consumption since a CPU is more power demanding than a DMAC [4], [5], [44].

Since DMACs can access memory, where critical data and code are located, they can be used by attackers to overtake or crash the system. Examples include abusing a GPU DMAC to gain privilege escalation [9] and a network interface DMAC to crash Linux [10]. To prevent DMAC attacks, many formally verified high-security hypervisors and operating systems [23]–[30] either disable DMACs or rely on IOMMUs (memory management units [15]–[17] placed between the DMAC and memory). The use of IOMMUs have three significant disadvantages: not all hardware platforms have IOMMUs; it negatively impacts performance and further reduces time predictability (due to additional translation table walks [18], [19]); and it requires additional non-trivial (potentially buggy [20]–[22]) software for configuring and protecting page tables and associated data structures.

Work partially supported by the TrustFull project financed by the Swedish Foundation for Strategic Research.

Verifying memory safety in presence of DMACs and absence of IOMMUs require formal models of the DMAC hardware including the interface between DMAC, software and memory. Such models allow reasoning about the effects of software accessing DMAC registers, of DMAC memory accesses, and the interaction between of software and DMAC which share data structures in memory.

We present a general framework for modeling DMACs (Section III). The framework is implemented in the HOL4 interactive theorem prover [31] and includes a general DMAC model which can be instantiated to a given DMAC by defining 14 DMAC specific functions (the most significant ones are listed in Table II). This generalization allows us to identify and verify sufficient conditions to confine DMAC memory accesses to certain memory regions.

To achieve this general verification result, in Section IV we establish a refinement between an abstract DMAC model, which is easier to analyze, and identify sufficient conditions to preserve the refinement that must be satisfied by the DMAC instantiation and the DMAC driver. This strategy has three main benefits: (1) the refinement theorem can be reused to verify functional correctness of drivers using the abstract model; (2) the verification of the instantiation deals only with the identified sufficient conditions and do not have to deal with the entire transition system of the DMAC model; and (3) the software conditions can be verified using the abstract model.

In order for the framework to be as general as possible, we have reviewed numerous DMACs (Table I). In Section V we demonstrate our approach by instantiating the framework with the USB DMAC in an SoC from Texas Instruments [32]. We use our result to identify the conditions that must be satisfied by a driver or a security monitor. The use of the framework has largely reduced the time for analyzing the USB DMAC.

Finally, in Section VI we discuss the HOL4 implementation and the security analysis of the Linux USB DMAC driver.

II. Background

DMACs perform memory accesses by operating on a queue of buffer descriptors (BDs), illustrated in Fig. 1, which are initialized by the driver. Each BD contains information about

https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_18

This article is licensed under a Creative Commons Attribution 4.0 International License

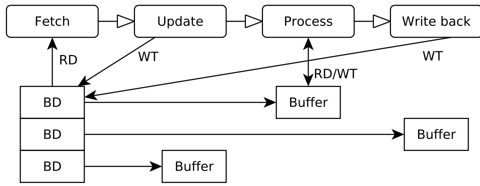

Fig. 1. DMAC

a memory transfer and the status of that transfer. The queue can be stored either in internal DMAC memory or in external main memory, either as a linked list (potentially cyclic), a ring, or as an array. Once the driver has initialized the BDs, the driver signals the DMAC to start operating on the BDs in the queue, which is done by a pipeline consisting of four stages. **(1) Fetch**: The DMAC fetches the BD into internal CPU-inaccessible memory. **(2) Update**: If the BD is operated on in multiple rounds, then the DMAC updates the BD to reflect the remaining transfers to perform for subsequent rounds. **(3) Process**: The DMAC performs the direct memory accesses (DMA transfers) to the buffers in main memory as specified by the BD. **(4) Write back**: If all memory accesses specified by the BD have been performed, the DMAC writes back the BD to signal the driver that the BD has been processed and can be reused for new transfers.

In the following we use $\mathcal{O} = \{f, u, p, w\}$ to refer to these four operations. The DMAC may also perform memory accesses due to **maintenance** operations, for example to store statistics or management data in memory. These operations are not atomic and may require multiple memory accesses. Furthermore, DMACs may be able to work on multiple queues of BDs concurrently, where each queue constitutes one DMA channel, and each channel may have more than one BD in each of its pipeline stages.

Both the driver and the DMAC can read and modify the queues: The driver reads the status of existing BDs and appends new BDs; the DMAC reads and updates BDs. For this reason verifying properties of this kind of system is challenging and similar to verifying concurrent threads sharing memory. In order to control the complexity caused by the interleaving of these the CPU/driver and the DMAC, the verification must exploit some sort of rely/guarantee [6], that enables verification of each component in isolation while assuming properties of the other component. Our verification approach follows this strategy, showing that there are sufficient conditions (rely) that if met by the driver allow to restrict (guarantee) the memory accesses of the DMAC.

A. DMAC Characteristics

In order to support a wide range of DMACs, our general model must accurately describe the memory accesses that may be performed by an arbitrary DMAC. To identify the common features of DMACs, we studied eight stand-alone DMACs, six embedded in USB controllers, and five embedded in Ethernet controllers, and the DMAC of IBM Cell, some characteristics of which are listed in Table I. The main difference among the DMACs is the mechanism used to organize BD queues: 13 DMACs use linked lists; five use ring buffers; and two use queues of one single BD. Moreover, seven DMACs store the queues in internal memory and 13 store the queues in main memory; Furthermore, DMACs have different: internal states (e.g., address pointers, counters, and state machines); number of DMA channels; reactions to register accesses made by the CPU; scheduling of channels; BD format (e.g. fields for buffer start address and size); and behavior of the four pipeline stages (fetch, update, process, and write back).

Stand-alone DMACs		
Chip	BD Organization	BD Location
Texas Instruments AM335x	Linked list	Internal memory
Microchip PIC32 Family	Linked list	Internal memory
Xilinx AXI DMA v7.1	Linked list	Main memory
NXP MPC5675/KMPC57xx	Linked list	Internal memory
Infineon GPDMA	Linked list	Main memory
Broadcom BCM2835	Linked list	Main memory
ST Microelectronics STR91xFA	Linked list	Main memory
Texas Instruments TMS320C5515	Linked list	Main memory
IBM Cell BE	Array/Ring	Main memory
USB DMACs		
Chip	BD Organization	BD Location
Cypress EZ-USB FX3	Linked list	Main memory
Xilinx Zyng-7000	Linked list	Main memory
Texas Instruments AM335x	Linked list	Main memory
NXP SAF1761 USB OTG	One BD	Internal memory
STM32F72xxx/STM32F73xxx	One BD per channel	Internal memory
Microchip PIC32 Family	Ring	Main memory
NIC DMACs		
Chip/Board	BD Organization	BD Location
Texas Instruments AM335x	Linked list	Internal memory
Broadcom NetXtreme/Netlink	Ring	Main memory
Realtek Ethernet RTL8100	Ring	Internal memory
3Com 3C90x/B	Linked list	Main memory
Intel e1000/e, X550, I350, I210	Ring	Main memory

TABLE I
STUDIED DMACS.

B. Security Threat from DMACs

Without an IOMMU, a DMAC can access memory without restrictions. For instance, consider a microkernel (or a hypervisor), where a user-mode driver (or a guest) should not be able to directly access kernel memory. If the driver can directly configure a DMAC that can perform memory-memory transfers, then the driver could store a malicious program in its own memory, and configure the DMAC to transfer this buffer to the exception handling table of the kernel. This results in code injection, bypassing the normal protection provided by the MMU that prevents direct tampering from the driver. Similarly, the driver of an Ethernet controller may overwrite kernel data structures with an incoming network packet or to leak data in kernel memory.

In order to isolate a DMAC, its configuration must meet three sufficient conditions, which are all violated by the example of Fig. 2:

1) BDs specify DMA reads and writes to buffers that are considered “readable” and “writable”: BD1 can instruct the DMAC to violate isolation since part of the buffer is outside the allowed memory region.

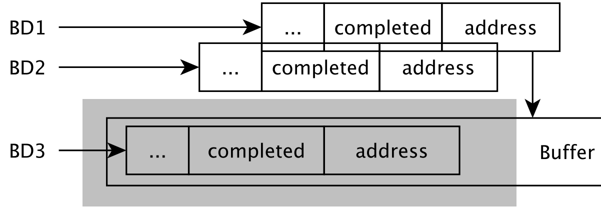

Fig. 2. DMAC isolation violations. Readable and writable region is colored in gray.

2) If BDs are stored in main memory, then the BDs must be located in "readable" and "writable" memory and must not specify DMA writes to BDs: The DMAC will violate memory isolation when fetching, updating and writing back BD1. Also, the DMAC can modify BD3 while processing BD1, since BD3 overlaps the buffer addressed by BD1.

Basically, these conditions guarantee that the BDs "instruct" the DMAC to access only "readable" and "writable" memory, and that the DMAC cannot change such BD "instructions".

III. General DMAC Model

We assume a computer system to be the composition $c|m|d$, where each component represents the state of a CPU, a memory and a DMAC respectively. We use standard synchronous composition of the transition systems of the components (assuming that parallel composition is associative, symmetric, and commutative):

$$\frac{x \xrightarrow{\tau} x'}{x|y \xrightarrow{\tau} x'|y} \qquad \frac{x \xrightarrow{l} x' \qquad y \xrightarrow{\bar{l}} y'}{x|y \xrightarrow{\tau} x'|y'}$$

The labels of these transition systems are τ for internal operations, and $rd(as, bs)$/$wt(as, bs)$ for reading/writing the bytes bs at/to the locations with addresses as, where the latter two have co-labels $\overline{rd(as, bs)}$ and $\overline{wt(as, bs)}$.

We do not explicitly define the CPU model. This model could for instance be the formalization of an Instruction Set Architecture (ISA) or a more abstract model of a device driver. Memory is an array of bytes, where $\mathcal{M}$ represents the addresses of the main memory:

$$\frac{as \subseteq \mathcal{M}}{m \xrightarrow{\overline{rd(as, m[as])}} m} \qquad \frac{as \subseteq \mathcal{M}}{m \xrightarrow{\overline{wt(as, bs)}} m[as \mapsto bs]}$$

Notice that we use early semantics: the memory is always ready to receive a memory update non-deterministically selecting all possible bytes bs. This non-determinism is resolved when the the memory transitions system is composed with another transition system that performs a write.

A. DMAC Transition System

The DMAC state consists of three components, $d = (s, b, c)$: An internal state s, whose type depends on the specific DMAC; a messsage box b containing memory requests and replies; and a DMA channel c (the model supports multiple channels, but we omit them for simplicity). We will use Fig. 3 to illustrate the model, where a queue of five BDs has been configured in main memory and each BD points to a buffer.

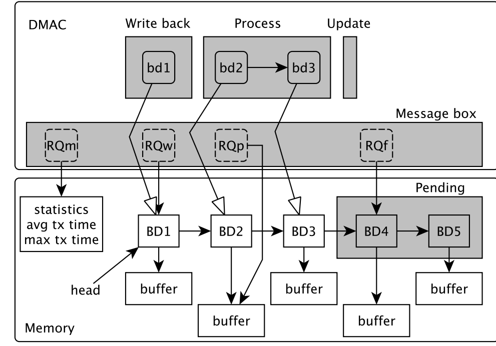

Fig. 3. DMAC model.

The message box b allows the DMAC to operate asynchronously w.r.t. the memory. This box is a set of memory read and write requests and replies: $\mathbf{r}_t^{op}[as]$, $\mathbf{w}_t^{op}[as, bs]$ and $\mathbf{p}_t^{op}[bs]$, where op, t, as and bs denote: The DMAC pipeline or mainteinance operation $\mathcal{O} \cup \{m\}$ that issued the memory request or that shall have the reply; a memory request-reply identifier tag; addresses to read/write; and bytes read/written.

The component $c : \mathcal{O} \setminus \{f\} \hookrightarrow \mathcal{B}$ models the DMAC pipeline. In the following we use $c.op$ to denote $c(op)$. Hence, $c.u = [bd_1, ..., bd_n]$ denotes the queue of BDs in the update stage, with n arbitrary and $n = 0$ denoting an empty queue; and similarly for p and w. We call these abstract BDs, since they are records whose type depends of the specific DMAC and contain the same information that is stored by the BDs in main memory. Independently of the DMAC instantiation, a BD bd always contains four mandatory fields specifying the addresses of the locations: where it is stored $bd.ra$, that are updated when it is written back $bd.wa$ (e.g., the address of its completion flag), and of the buffer that must be read and written via DMA, $bd.dra$ and $bd.dwa$. The BDs in c are the ones that have been fetched with each BD being in some DMAC pipeline stage. For instance, in Fig. 3 three BDs have been fetched and are therefore in the DMAC pipeline (bd2 and bd3 are being processed and bd1 is currently written back). We use "pending" BDs to refer to the BDs in the queue that are left to fetch (e.g. BD4 and BD5). Normally, the concatenation of the queues in c represents a sliding window of the queue in memory.

To account for the DMAC specifics the rules describing DMAC transitions are defined in terms of two records. The record Δ contains behavioral functions that model the specific actions of a DMAC. The record Π contains projection functions that extract information from the state and returns the proper data structures (e.g., BDs). These DMAC specific functions must be defined to obtain a concrete DMAC model. Table II summarizes the behavioral functions (except for a scheduler that resolves non-determinism) and the two most important projection functions.

Function	Modeled Operation/State Information
$\Delta.rr$	(See rule $[\boldsymbol{rr}]$) Given an internal state s_1 and the addresses as of the DMAC register to read, returns an updated internal state s_2, the read bytes bs, and potential maintenance memory requests rs associated with the read.
$\Delta.wr$	(See rule $[\boldsymbol{wr}]$) Given an internal state s_1, the addresses as of the DMAC register to write, the bytes bs to write, returns an updated internal state s_2 and potential maintenance memory requests rs associated with the write.
$\Pi.fas$	(See rule $[\boldsymbol{f_1}]$) Given an internal state s, returns the memory read request $\mathbf{r}_t^f[as]$ for fetching the next part of the BD being fetched at addresses as and with request identification tag t.
$\Delta.f$	(See rules $[\boldsymbol{f_2}]$ and $[\boldsymbol{f_3}]$) Given an internal state s_1, and for external BDs a fetch reply $\mathbf{p}_t^f[bs]$, (where the bytes bs constitutes a part of the currently fetched BD and with t being the request identification tag of the corresponding read request), but $\bot$ for internal BDs; returns an updated internal state s_2, and either a fetched BD bd and the stage $op \in \{u,p\}$ the BD shall be moved to, or $\bot$ if additional external or internal memory reads are necessary to fetch the next BD.
$\Delta.p$	(See rule $[\boldsymbol{p_t}]$) Given an internal state s_1, the first BD bd in the process stage and whose memory transfers are currently being performed (i.e., the DMA transfers specified by bd), and the DMA read replies ps associated with the process stage; returns an updated internal state s_2 reflecting the processing of the given memory replies and the generatation of potentially new memory requests rs, and a boolean flag indicating whether all requests/replies associated with bd have now been issued/processed and the BD shall be moved to the write back queue.
$\Delta.w$	(See rule $[\boldsymbol{w}]$) Given an internal state s_1, and the BDs in the write back queue $c.w$; returns an updated internal state s_2, the memory write requests rs containing the bytes to write to memory associated with any given BD (not used for internal BDs), and the BDs bds that are now released due to the write back (removed from the write back queue).
$\Delta.m$	(See rule $[\boldsymbol{m}]$) Given an internal state s_1 and memory read replies ps (to read requests issued by $[\boldsymbol{rr}]$ and $[\boldsymbol{wr}]$); returns an updated internal state s_2 and the processed replies pps that shall be removed from the message box.
$\Pi.cf$	(See rules $[\boldsymbol{w}]$ and $[\boldsymbol{ma}]$ in Subsection IV-A) Given internal state s and memory m, returns the pending BDs bds that remains to fetch (bds = [BD4, BD5] in Fig. 3).

TABLE II
SUMMARY OF THE DMAC SPECIFIC FUNCTIONS.

In the following we use $\mathcal{D}$ to represent the set of addresses of DMAC registers. The reaction of the DMAC when the CPU accesses such a register at addresses as is DMAC specific and must be described by the Read Register and Write Register functions: $\Delta.rr$ and $\Delta.wr$. Notice that these functions can affect the internal state of the DMAC and may return memory requests rs in case a register access makes it necessary for the DMAC to update maintanence data in main memory ($c = a + b$ denotes $c = a \cup \{b\} \wedge a \cap \{b\} = \emptyset$):

$$\frac{(s_2, bs, rs) = \Delta.rr(s_1, as) \qquad as \subseteq \mathcal{D}}{(s_1, b, c) \xrightarrow{\overline{rd(as,bs)}} (s_2, b + rs, c)} [\boldsymbol{rr}]$$

$$\frac{(s_2, rs) = \Delta.wr(s_1, as, bs) \qquad as \subseteq \mathcal{D}}{(s_1, b, c) \xrightarrow{\overline{wt(as,bs)}} (s_2, b + rs, c)} [\boldsymbol{wr}]$$

The message box acts as a buffer between the memory and the DMAC. The message box synchronizes with memory, consuming a request (previously produced by operation op and with identifier t) and for reads adding a correponding reply:

$$(s, b + \mathbf{r}_t^{op}[as], c) \xrightarrow{rd(as,bs)} (s, b + \mathbf{p}_t^{op}[bs], c) \quad [\boldsymbol{rm}]$$

$$(s, b + \mathbf{w}_t^{op}[as, bs], c) \xrightarrow{wt(as,bs)} (s, b, c) \quad [\boldsymbol{wm}]$$

The other rules are for internal DMAC transitions. For fetching BDs ($op = f$) there are five cases: three if BDs are stored in main memory and two if BDs are stored in internal memory. $[\boldsymbol{f_1}]$ describes the first step in fetching an external BD, that is applicable when there are no pending memory replies for BD fetches. In this case a memory request is added to the message box for fetching new BDs. In Fig. 3, the rule can produce the request RQf when starting to fetch BD4. The addresses and the tag are given by the function Fetch Addresses $\Pi.fas$:

$$\frac{\{\mathbf{p}_{t'}^f[bs] \in b\} = \emptyset \qquad \mathbf{r}_t^f[as] = \Pi.fas(s)}{(s, b, c) \xrightarrow{\tau} (s, b + \mathbf{r}_t^f[as], c)} [\boldsymbol{f_1}]$$

When a memory read request for fetching a BD is served, the corresponding reply is added to the message box. $[\boldsymbol{f_2}]$ describes the behavior when such a reply exists but more reads are necessary to fetch the complete BD, in which case the function Fetch $\Delta.f$ returns $\bot$. $\Delta.f$ can update the internal state with the consumed reply, which contains a partial BD:

$$\frac{(s_2, \bot) = \Delta.f(s_1, \mathbf{p}_t^f[bs])}{(s_1, b + \mathbf{p}_t^f[bs], c) \xrightarrow{\tau} (s_2, b, c)} [\boldsymbol{f_2}]$$

$[\boldsymbol{f_3}]$ handles the case when a BD fetch reply $\mathbf{p}_t^f[bs]$ exists and it contains the last chunk of bytes bs of the BD bd being fetched. In this case $\Delta.f$ returns a pair consisting of the abstract representation of the fetched BD bd and which pipeline stage queue $op \in \{u, p\}$ the BD shall be appended to (denoted by $+\!\!+$):

$$\frac{(s_2, (bd, op)) = \Delta.f(s_1, \mathbf{p}_t^f[bs])}{(s_1, b + \mathbf{p}_t^f[bs], c) \xrightarrow{\tau} (s_2, b, c[op \mapsto c.op +\!\!+ bd])} [\boldsymbol{f_3}]$$

The fetching BD rules for DMACs with internal BDs are similar to $[\boldsymbol{f_2}]$ and $[\boldsymbol{f_3}]$, but no memory requests and replies are involved, since BDs are obtained from the internal DMAC state.

Two rules model the process stage ($op = p$), depending on whether the currently processed BD is now completed or not. The following rule covers the case when a BD is completely processed (the other case when more DMA transfers remain of the BD is similar, but keeps the BD at the head of the process queue). In either case, the function Process $\Delta.p$ models the DMAC specific behavior of generating and processing memory requests and replies. It takes the currently processed BD bd at the head of $c.p$, and pending memory replies for the process stage; and returns an updated internal state, optional new memory requests rs, and a completion flag which specifies if the BD has now been processed and shall be moved to the write back stage. These requests represents DMA

reads and writes, while the replies are the results of previously issued read requests that have been served by memory. All replies are consumed and the new requests are added to the message box. In Fig. 3 the rule can produce the request RQp to write the buffer addressed by bd2.

$$\frac{c.p = bd :: bds \quad ps = \{\mathbf{p}_t^p[bs] \in b\} \qquad (s_2, rs, \mathbf{true}) = \Delta.p(s_1, bd, ps)}{(s_1, b, c) \xrightarrow{\tau} (s_2, b - ps + rs, c[p \mapsto bds, w \mapsto c.w +\!\!+ bd])} [\boldsymbol{p_t}]$$

Updating and writing back BDs are similar and for this reason we only describe write back in detail. The main difference is that updating a BD moves the updated BD from the head of update queue to the tail of the process queue, while a write back may remove a (possibly empty) prefix of BDs from the write back queue $c.w$. If BDs are stored in main memory, the Write back function $\Delta.w$ returns the memory write requests rs for writing back the BDs, while internal BDs are written back by updating the internal state (in Fig. 3 the rule can produce the request RQw to update bd1 in main memory):

$$\frac{(s_2, rs, bds) = \Delta.w(s_1, c.w)}{(s_1, b, c) \xrightarrow{\tau} (s_2, b + rs, c[w \mapsto c.w - bds])} [\boldsymbol{w}]$$

Finally, the DMAC can react to the replies ps to the read requests produced by the mainteinance operations (i.e., requests issued by $[\boldsymbol{rr}]$ and $[\boldsymbol{wr}]$), removing the processed replies $pps \subseteq ps$ from the message box:

$$\frac{ps = \{\mathbf{p}_t^m[bs] \in b\} \qquad (s_2, pps) = \Delta.m(s_1, ps)}{(s_1, b, c) \xrightarrow{\tau} (s_2, b - pps, c)} [\boldsymbol{m}]$$

IV. Verification

Our goal is to verify general conditions that are sufficient to guarantee DMAC isolation (Theorem 1): The DMAC can only read "readable" and write "writable" memory regions, denoted by the sets of addresses $\mathcal{R}$ and $\mathcal{W}$.

Our verification is based on refinement. Let M_3 be the DMAC model defined in Section III. We introduce two layered abstractions M_2 and M_1. For each model M_{i+1} we introduce an invariant $\mathcal{I}_{i+1}$ that allows us to prove bisimulation between M_{i+1} and M_i. We finally introduce an invariant $\mathcal{I}_1$ for M_1 that demonstrate DMAC isolation and use the bisimulation to transfer this property down to the M_3 DMAC model. This strategy has three benefits: (i) it allows us to solve one problem at a time via a single refinement step; (ii) it establishes a bisimulation between the concrete model and the more abstract one, which allows further properties (e.g., functional correctness of a device driver) to be verified using abstract models; (iii) it allows us to identify assumptions that all DMAC instantiations and drivers must satisfy in the form of proof obligations. The obligations must be proved for a given DMAC instantiation, but these proofs depend only on the instantiation (Δ and Π) in contrast to a complete DMAC model. The driver conditions can be proven relying only on the DMAC guarantee that are established by our verification.

A. Abstract DMAC Models

The lower abstraction M_2 is a virtual DMAC that cannot self-modify pending BDs. This property allows a driver to prepare, extend, and read the queue that must be fetched by the DMAC without being concerned that the DMAC may alter the queue. This is done by checking that pending BDs are not addressed by BD updates, write backs, and DMA writes. For instance, the rule for write back becomes (where $a \not\between b$ means that sets a and b are disjoint: $a \cap b = \emptyset$):

$$\frac{(s_2, rs, bds) = \Delta.w(s_1, c.w) \quad \left(\bigcup_{bd \in bds} bd.wa \cup \bigcup_{\mathbf{w}_t^{op}[as,bs] \in rs} as \right) \not\between \bigcup_{bd \in \Pi.cf(m,s_1)} bd.ra}{(s_1, b, c) \xrightarrow{\tau} (s_2, b + rs, c[w \mapsto c.w - bds])} [\boldsymbol{w}]$$

The rule prevents write backs from modifying pending BDs, independently of whether the BDs are stored in internal or main memory. For internal BDs, the locations modified by $\Delta.w$ are identified from the list of released BDs bds. For external BDs, the addresses are in the requests rs produced by $\Delta.w$. $\Pi.cf$ returns the list of remaining (Concrete) pending BDs to Fetch, as identified by the internal state and memory (BD4 and BD5 in Fig. 3).

The upper abstraction M_1 guarantees that BDs cannot be changed by the CPU. The pending BDs to fetch are stored in an abstract queue $c.f$. By definition the CPU cannot modify or remove entries from this list, but it can append BDs by either: writing a DMAC register (e.g. by writing the tail pointer register or by writing the next pointer field of a BD in external memory). This makes it possible to prove properties of DMA transfers (e.g., memory isolation) without considering interleavings with CPU transitions which can potentially corrupt pending BDs. This abstract model alters the previous transition system by composing the abstract DMAC and memory in such a way that the abstract DMAC can "magically" extend the abstract queue of pending BDs with new BDs bds when the CPU writes memory m_1 at locations with addresses as and bytes bs resulting in memory m_2 (writing registers is similar but with the updated internal state considered instead of updated memory):

$$\frac{as \subseteq \mathcal{M} \qquad m_2 = m_1[as \mapsto bs] \quad bds' = \Pi.cf(m_2, s) \qquad \exists bds.\ bds' = c.f +\!\!+\ bds}{m_1|(s, b, c) \xrightarrow{\overline{wt(as,bs)}} m_2|(s, b, c[f \mapsto bds'])} [\boldsymbol{ma}]$$

The internal operations of M_1 also differ. For $[\boldsymbol{f_3}]$, the BD bd returned by $\Delta.f$ is ignored and instead the first BD of $c.f$ is moved to $c.u$ or $c.p$, depending on whether the BD shall be updated or not. The reason why main memory is still accessed to fetch BDs (even though they are not used) is to keep the transition systems synchronized: Internal states are updated identically in both M_1 and M_2. In addition, the checks for updates/write backs and DMA writes in M_2 are also in M_1.

B. Refinement Relations, Invariants, and Proof Obligations

We use $(m, d_{i+1}) \simeq_{i+1} (m, d_i)$ for the refinement relation between M_{i+1} and M_i. These relations require the common

state fields to be equal: $d_{i+1} = d_i$. Additionally $(m, d_2) \simeq_2 (m, d_1)$ requires that the abstract and concrete pending BDs are equal: $d_1.ch.f = \Pi.cf(m, d_2.s)$.

The refinement proofs depend on invariants that restrict the state of the lower layer. The invariant for M_2 requires that no DMA write request targets pending BDs $\mathcal{I}_2(m, s, b, c) \coloneqq$

$$\mathbf{w}_t^{op}[as, bs] \in b \wedge bd \in \Pi.cf(m, s) \implies as \not\asymp bd.ra$$

This invariant simply propagates the checks of the internal abstract DMAC operations (e.g., $[\boldsymbol{w}]$ of M_2).

In order to establish the bisimulation for the model of Section III, we also need an invariant that enforces the same constraints that are checked by the abstract models. The invariant $\mathcal{I}_3$ requires that every pending or fetched BD in the pipeline do not have update/write back addresses nor DMA writes to pending BDs (this includes that pending BDs do not overlap; in the definition of $\mathcal{I}_3$, c denotes the concatenation of $c.u$, $c.p$ and $c.w$): $\mathcal{I}_3(m, s, b, c) \coloneqq$

$$\bigcup_{bd \in c \cup \Pi.cf(m,s)} (bd.wa \cup bd.dwa) \not\asymp \bigcup_{bd \in \Pi.cf(m,s)} bd.ra$$

The last invariant restricts M_1 to force the DMAC to access only readable and writable memory (in the definition of $\mathcal{I}_1$, c denotes the concatenation of $c.f$, $c.u$, $c.p$ and $c.w$): $\mathcal{I}_1(m, s, b, c) \coloneqq$

$$\bigcup_{\mathbf{r}_t^{op}[as] \in b} as \cup \bigcup_{bd \in c} bd.ra \cup \bigcup_{bd \in c.op, op \neq w} bd.dra \subseteq \mathcal{R} \;\wedge$$

$$\bigcup_{\mathbf{w}_t^{op}[as,bs] \in b} as \cup \bigcup_{bd \in c} bd.wa \cup \bigcup_{bd \in c.op, op \neq w} bd.dwa \subseteq \mathcal{W}$$

The instantiation of a given DMAC must satisfy some proof obligations, which mainly state that the behavioral and projection functions are consistent:

1) A fetched BD (by $[\boldsymbol{f_3}]$) is the first pending BD: If $\mathbf{r}_t^f[as] = \Pi.fas(s_1)$, and $(s_2, (bd, op)) = \Delta.f(s_1, \mathbf{p}_t^f[m[as]])$, then there exist BDs bds such that $\Pi.cf(m, s_1) = bd :: bds$. Also, after fetching a BD, the projection function must reflect the removal of the BD from the pending queue: $\Pi.cf(m, s_2) = bds$.
2) The queue of pending BDs depends only on the locations of the BDs and the internal state: If $\forall a \in \bigcup_{bd \in \Pi.cf(m,s)} bd.ra.\ m_2[a] = m_1[a]$, then $\Pi.cf(m_1, s) = \Pi.cf(m_2, s)$.
3) The function associated with DMA transfers does not affect the queue of pending BDs: $(s_2, rs, cf) = \Delta.p(s_1, bd, ps)$ implies $\Pi.cf(m, s_2) = \Pi.cf(m, s_1)$

The proof obligations of the driver are that it only appends BDs and preserves the invariants $\mathcal{I}_1$ and $\mathcal{I}_3$. This proof obligation is only relevant for non-internal CPU transitions, since the invariants do not depend on the CPU. For memory writes (other cases are similar) this means that if $\bigwedge_{i \in \{1,3\}} \mathcal{I}_i(m, d)$, $cpu \xrightarrow{wt(as,bs)} cpu'$, and $as \subseteq \mathcal{M}$ then:

1) $\exists bds.\ \Pi.cf(m[as \mapsto bs], d.s) = \Pi.cf(m, d.s) +\!\!+ bds$.
2) $(m|d) \xrightarrow{\overline{wt(as,bs)}}_1 (m'|d')$ implies $\mathcal{I}_i(m', d')$, where $\rightarrow_1$ denotes the transition relation of M_1.

That is, writes (updates, write backs and DMA writes) of appended BDs do not point to pending BDs or non-writable memory, appended BDs do not overlap, and reads (both fetches and DMA) of appended BDs do not point to non-readable memory. Notice that invariant preservation can be done by checking the state of the the more abstract DMAC model M_1, disregarding the lower layers.

C. Refinement and Memory Isolation

Refinement is phrased as a bisimulation and assumes the invariant. For $i \in \{2, 3\}$ ($\rightarrow_i$ denotes the transition relation of M_i):

Lemma 1. *If $\mathcal{I}_{i+1}(m, d)$, $(m, d) \simeq_{i+1} (m, e)$, and $(c, m, d) \xrightarrow{l}_i (c', m', d')$ then exists e' such that $(c, m, e) \xrightarrow{l}_i (c', m', e')$ and $(m', d') \simeq_{i+1} (m', e')$, and vice versa with transitions of $\rightarrow_i$.*

Proof. Consider $i = 1$. For the fetch rules the main difference between M_1 and M_2 is that M_1 fetches abstract BDs and M_2 fetches concrete BDs. $\simeq_2$ guarantees that these queues are equal. M_1 moves the first BD of $d_1.c.f$ to the tail of the update or process queue ($d_1.c.op$, $op \in \{u, p\}$). DMAC proof obligation 1) ensures that M_2 performs a corresponding operation by moving the first concrete BD of $\Pi.cf(m, d_2.s)$.

For updating, processing and writing back BDs, the abstract pending BDs of M_1 cannot change by definition. To show that the concrete pending BDs of M_2 are also unchanged we use the the update/write back checks in M_2 and DMAC proof obligation 2). Moreover, $\mathcal{I}_2$ and DMAC proof obligation 2) imply that memory writes do not change concrete pending BDs in M_2, preserving equality between concrete and abstract BDs queues.

Finally, for CPU transitions, there are two cases depending on whether the pending BDs are modified. If not, then memory and register accesses have identical effects in M_1 and M_2. Otherwise, Driver proof obligation 1) ensures that M_2 only appends BDs. This allows M_1 to produce the corresponding abstract queue of pending BDs by extending the existing one via the rule $[\boldsymbol{ma}]$ (and similarly for register writes).

For $i = 3$, $\mathcal{I}_3$ is transferred by $\simeq_3$ to M_2, implying that all checks in M_2 pass (e.g. $[\boldsymbol{w}]$). Thus, M_2 and M_3, perform identical operations. CPU and DMAC memory transitions are identical in M_2 and M_3. □

We then prove that invariants are preserved and transfered by the refinements:

Lemma 2. *If $\mathcal{I}_i(m, d)$ and $(c, m, d) \xrightarrow{l}_i (c', m', d')$ then $\mathcal{I}_i(m', d')$. Also if $j < i$ and $(m, d_j) \simeq_j (m, d_{j+1})$ then $\mathcal{I}_i(m, d_{j+1}) \Leftrightarrow \mathcal{I}_i(m, d_i)$.*

Finally we show that DMAC transitions modify and depends on only the right regions of memory (where $f|_A$ is the projection of a function over domain A and $\bar{A}$ is set complement):

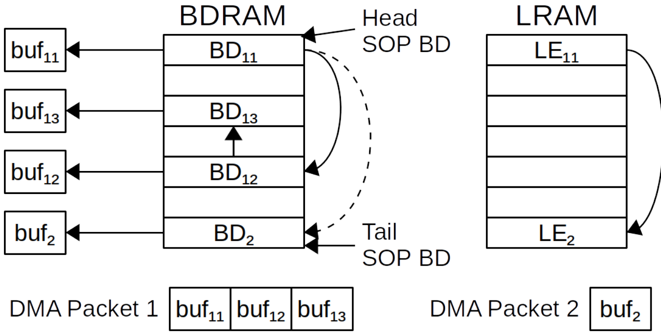

Fig. 4. Organization of BD queues of the USB DMAC.

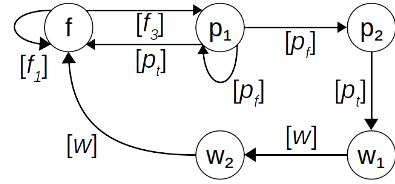

Fig. 5. State diagram of the USB DMAC instantiation. The transition labels denote the rules that cause the corresponding transition.

Theorem 1. *If* $\bigwedge_i \mathcal{I}_i(m,d)$ *and* $(c,m,d) \rightarrow_3 (c,m',d')$ *then* $m|_{\bar{\mathcal{W}}} = m'|_{\bar{\mathcal{W}}}$*, and if* $m|_{\bar{\mathcal{R}}} = m_1|_{\bar{\mathcal{R}}}$ *then* $(c_1,m_1,d) \rightarrow_3 (c_1,m_1',d')$ *and* $m'|_{\bar{\mathcal{W}}} = m_1'|_{\bar{\mathcal{W}}}$

The theorem follows from Lemmas 1, 2 and by establishing a further bisimulation with an even more abstract layer that is isolated by construction. This model have additional checks compared to M_1 that prevent adding memory requests to the message box that point outside $\mathcal{R}$ and $\mathcal{W}$.

V. USB DMAC

We instantiate our framework with the DMAC of the USB controller of the AM3358 SoC by Texas Instruments [32], the SoC on the development board BeagleBone Black [7]. As Fig. 4 illustrates, the DMAC organizes BD queues by means of two memory regions, one storing BDs (BDRAM) and one storing linking information (LRAM), the base addresses of which are configurable. Both regions are organized as arrays with the same number of entries. To transmit a DMA packet, potentially scattered in memory in multiple buffers (e.g., DMA packet 1 is the concatenation of buf_{11}, buf_{12} and buf_{13}), the driver initializes in BDRAM one BD for each buffer (BD_{11}, BD_{12} and BD_{13}), linking them via the next descriptor pointer in the order the data buffers shall be transmitted to the USB device. The first BD of a packet is called Start Of Packet (SOP). The LRAM is used to link packets: if BDRAM$[i]$ is a SOP then LRAM$[i]$ links the SOP BD of the next DMA packet (BD_{11} is linked to BD_2 via LRAM entry LE_{11}, in effect linking DMA packets 1 and 2). Both the driver and the DMAC read and write BDRAM, but only the DMAC uses LRAM.

To enqueue a DMA packet the driver writes the address of its SOP BD (e.g., BD_2 to enqueue packet 2) to the enqueue register `Q`. This write causes the DMAC to append the BDs of the new DMA packet to the pending queue: The LRAM entry of the previous tail SOP BD (e.g., LE_{11}) is updated with a link to the appended SOP BD. Once a BD has been fetched, it is processed, without being updated, and finally written back. A write back moves the head SOP BD of the transferred DMA packet from the pending queue to the tail of the completion queue (which is another queue whose links are also stored in LRAM). The completion queue is traversed by the driver to recycle BDs. The driver does this by reading the `C` register, making the DMAC return the address of the first SOP BD in the completion queue, and read LRAM to find the next completed SOP BD which now becomes the first SOP BD in the completion queue.

We focus on the instantiation of the transmission channel, since reception is similar. The internal state is a record $s = (r, hp, tp, hc, tc, t)$ containing the registers r (except `Q` and `C` which are not physical registers), the head and tail pointers of the pending and completion queues hp, tp, hc, tc, and the state t of the automaton in Fig. 5 that keeps track of the state of the operation of the current DMA packet in transfer.

In state `f`, the rules $[\boldsymbol{f_1}]$ and $[\boldsymbol{f_3}]$ fetch the next BD and move it to the process queue $c.p$ (BDs are fetched atomically, making $[\boldsymbol{f_2}]$ unnecessary; thus, $\Delta.f$ always returns a BD).

In state $\texttt{p}_1$, $[\boldsymbol{p_f}]$ repeatedly obtains memory read requests and handles replies until all data in the buffer has been read. If the BD in $c.p$ is not the last BD of the DMA packet (e.g. BD_{12}) then $[\boldsymbol{p_t}]$ sets the next state to `f` to operate on the next BD of the DMA packet (e.g. BD_{13}). Otherwise, after processing the last byte of the buffer, a further application of $[\boldsymbol{p_f}]$ is used to produce a DMA read request needed to read the LRAM entry of the SOP BD (LE_{11}) of the DMA packet in transfer. This data is needed later to update the linking ram in the write back stage and must be read by $[\boldsymbol{p_t}]$, since $[\boldsymbol{w}]$ cannot read memory. The state is set to $\texttt{p}_2$, in which $[\boldsymbol{p_t}]$ processes the reply containing the LRAM entry and sets the state to $\texttt{w}_1$.

Write backs are performed in two steps. First, in state $\texttt{w}_1$, $[\boldsymbol{w}]$ updates the head pointer hp of the pending queue to the address of the next SOP BD (BD_2), which has been previously retrieved in $\texttt{p}_2$. Second, in state $\texttt{w}_2$, the tail pointer tc of the completion queue is set to the address of the completed SOP BD (BD_{11}); the LRAM entry of the previous tail SOP BD of the completion queue is now linked to the new tail (completed) SOP BD (e.g. BD_{11}); the next state is `f` to fetch the next SOP BD (BD_2); and all BDs accumulated in $c.w$ are released, meaning that the driver can reuse them.

Register accesses are performed by directly reading and writing $s.r$, except `Q` and `C`. When `Q` is written, tp is updated to the written address of the appended SOP BD. When `C` is read, the value of hc is returned with hc set to the address of the next SOP BD in the completion queue. These register accesses cause additional DMA management accesses to LRAM in order to reflect the queue updates (e.g., linking LE_{11} to LE_2 when BD_2 is written to `Q`).

The following is a description of $\Pi.cf$, and why $\Pi.cf$, $\Delta.f$ and $\Pi.\mathit{fas}$ satisfy DMAC proof obligation 1). $\Pi.cf(m,s)$ finds

Aspect	NIC DMAC w/o fw	USB DMAC w/ fw
LoC model	1500	2000
LoC verification	55000	2000
Modeling time	3 person-months	2 person-months
Verification time	9 person-months	1/2 person-month

TABLE III
EFFORT OF VERIFYING MEMORY ISOLATION OF A NIC [33] AND A USB DMAC WITH AND WITHOUT THE FRAMEWORK. THE HOL4 EXPERIENCE BEFORE THE NIC DMAC WORK WAS ABOUT FOUR MAN-MONTHS, AND ABOUT 30 MAN-MONTHS BEFORE THE USB DMAC WORK.

the unfetched BDs in four steps. (1) It retrieves the address of the *current SOP BD* of the current DMA packet in transfer. (2) The address of the next BD to fetch is obtained from hp. (3) If hp is zero, then the entire pending queue has been visited and the function returns the accumulated BDs so far. Otherwise, it collects the unfetched BDs of the current DMA packet, starting from the next BD to fetch and traversing the next descriptor pointer fields. (4) The next BD to fetch is the SOP BD of the next DMA packet, identified by reading the LRAM entry of the last visited SOP BD. The procedure continues with step 3. The first BD that is fetched by $\Delta.f$ is at the address given by $\Pi.fas$ obtained from hp, which is the address obtained by $\Pi.cf$ in step 2. Hence, the fetched BD is the first pending BD.

VI. APPLICATION AND EVALUATION

The framework consists of about 28000 lines of HOL4 code, including models and proofs. It was first described in pseudocode based on reviews of more than 20 DMACs, and then refined into HOL4 code [42]. The high-level design, definition, and proof took in total 18 person-months.

The instantiation of the USB DMAC consists of about 2000 lines for the model, and about 2000 lines for the proofs of the proof obligations. The model is based on the informal specification [32], which, as is common with informal specifications, contains undefined terms whose meaning must be derived from the (lacking) context, dispersed information, and typos. Similar to the framework, we started with high-level pseudocode that was gradually refined to remove ambuiguities and to make it fit the framework, requiring seven person-weeks. Verifying the proof obligations took about two additional person-weeks. In previous work [33], we have modeled and verified memory isolation of a NIC DMAC without the support of the frawemork, taking about three months of modeling and nine months in proving that the invariant is preserved. Due to significantly less time in using the framework, we believe that the framework provides significant assistance in verifying memory isolation of DMACs, with the main benefit being the proof of that the invariant is preserved. Table III makes a comparison between the efforts invested into verifying the NIC and USB DMACs with and without the framework.

The benefit of our approach is that we can establish soundness of the verification conditions independently of the driver. Then one can independently analyze the driver. For instance, the Linux driver of the USB DMAC uses only a limited set of the features of the device: It allocates one single BD per channel, meaning that the DMA packets consist of only one buffer, and it enqueues a new packet only after that the previous one has been completed. The driver allocates two memory regions for BDRAM and LRAM. These memory regions do not overlap, neither do the BDs, with each BD of each channel being allocated a fixed location. The Linux virtual memory manager allocates the BDRAM and LRAM regions, and likewise the DMA buffers for data transfers. Assuming that these memory regions are disjoint and located in "readable" and "writable" memory, this driver satisfies the two driver proof obligations as follows. First, the driver pops BDs from the completion queues by reading the `C` register, before reinitializing them and appending them by writing the `Q` register, thus only modifying the pending BD queues by appending BDs. LRAM is not accessed by the driver. Moreover, by assumption, BDs and DMA buffers are in readable and writable memory. The driver organizes the BDs in disjoint array slots in BDRAM, meaning that BDs do not overlap, and thus write back addresses do not coincide with read addresses of other BDs.

VII. RELATED WORK

Verification of Device Drivers without DMA Model checkers and interactive theorem provers have been used to verify various properties of drivers controlling devices without a DMAC: Reading from flash memory gives previously written data [34]; correct copying of data from memory to an ATAPI disk [35]; termination of a UART driver transferring data from memory to the external environment [36]; safety and liveness properties of a UART driver [37]; absence of data races and illegal memory accesses by a keyboard driver [38]; and equivalence between abstract and concrete models of an SPI driver and the SPI controller [39].

These devices do not have a DMAC, meaning that their memory isolation depends only on the memory accesses performed by the driver. For devices without a DMAC, methods have been investigated for synthesizing and (semi-) automatically generating device drivers that satisfy the interfaces of the OS and the I/O device [50], [51].

Hardware Verification Our work assumes that the the hardware implementation of the device satisfies its hardware-software interface. Hardware verification is indeed an ortogonal problem to the driver verification problem.

A DMAC is reminiscent of a CPU in the sense that BDs corresponds to instructions, BD operations correspond to an instruction pipeline, and concurrent DMA channels correspond to multiple instruction streams (threads) with BDs from different channels. These aspects have been investigated by the CPU formal verification community [52]–[54].

Specifically for DMAC implementations, Clarke et al. [40] have used model checking to verify that DMAC transfers are eventually completed, that the DMAC is eventually ready for new transfers, and that memory operations terminate. The analyzed DMAC is relatively simple: The DMAC maintains no queues nor multiple channels; its configuration depends only on the DMAC registers; and the next transfer can be programmed only after the previous transfer is finished. The

same DMAC design was later used to verify relationships between signals, including clock cycle delays [41].

Verification of DMAC Drivers Monniaux [43] has verified a USB driver that controls a DMAC, using a static C code analyzer designed to detect memory access and arithmetic errors. The driver and the device are modeled in C, with interleaved execution. The C analyzer can automatically verify that the driver and the controller access only allowed memory.

Even if an existing C analyzer largely automates verification, the framework addresses some of the limitations of this work. First, to automate the analysis, the C model coarsely overapproximates all possible device actions. In order to check soundness of this overapproximation, one should refine the model and prove some sort of refinement (see Subsection IV-C), which can be difficult in C and is not supported by the tool. Second, the use of a general C verification tool requires the model to be defined in terms of C semantics. For example, the tool is designed for 32-bit atomic variable accesses, but some devices may use single byte granularity. Third, it is not clear if the tool can analyze models of DMACs that have complex BD queues. In fact, the analyzed model has a relatively simple structure, where BD queues consist of three static arrays. Finally, the overapproximation used to automate the analysis may prevent it from being used to verify functional properties (e.g., a buffer is actually copied from source to destination), which the tool has no support for.

Donaldson et al. [46] have used model checking to verify absence of data races to DMA buffers between the PPE (a general CPU) and SPEs (HW accelerators) of the IBM Cell BE processor, which have embedded DMACs in the SPEs to transfer data between main memory and their local memory. In their analysis, BD queues are not considered, only single atomic DMA commands. Hence, this work is limited to this specific hardware and does not consider memory isolation.

Schwarz et al. [47] have used Coq to model a DMAC and a hypervisor, which virtualizes the DMAC among two guests, and verified that the DMAC virtualization keeps the guest isolated. Also this work concerns a specific and simple DMAC, not dealing with complex organizations of BD queues.

In previous work [33] we modeled a DMAC of an Ethernet NIC in HOL4 and verified sufficient conditions for isolating packets in transfer. The BD queues are organized as linked lists stored in internal DMAC memory. The formalization and verification took about one person-year, the majority of which can be saved with the DMAC framework.

Techniques for Isolating DMACs The ability of isolating DMA accesses is fundamental for guaranteeing security of entire systems. For instance, the security of several verified systems [23]–[30], [48], [49] requires restricted DMA.

Hardware assisted DMAC isolation uses stand-alone IOMMUs [15] or IOMMU embedded in the DMAC [8] to prevent the DMAC from accessing critical memory due to untrusted configurations. In absence of dedicated hardware mechanisms, the common approach to enforce memory isolation is via a monitor in the OS [44], [55] or the hypervisor [45], that intercepts driver reconfigurations of the DMAC. Other methods analyze an aspect of the system in runtime and react to violations: Execution of device firmware follows a pre-determined pattern [14] (e.g. the stack pointer and program counters are in valid memory regions), memory bus activity follows a pre-determined pattern [13], execution traces recorded by hardware or binary instrumentation [12], and integrity of firmware and I/O configuration (the checks of which are triggered by interrupts and thresholds of hardware performance counters) [11].

Grisafi et al. [56] presents a mechanism to isolate memory for low-end embedded systems with DMACs. This is achieved by means of a hypervisor, and a compiler that inserts hypervisor calls in applications accessing DMAC registers. The software design has been verified, however the security of the system depends on the fact that the security policies enforced by the hypervisor prevent the DMAC to access critical region of memory. While this is simple to check for simple DMACs with single BDs and that are configured only via memory mapped registers, guaranteeing this property for complex DMACs requires to analyze the device model. Our work is complementary to the software verification, since it supports the identification of the verification of the security policies for teh devices.

VIII. Conclusion

We have implemented a framework in the interactive theorem prover HOL4 for modeling DMACs, and by means of refinement formally verified DMAC memory isolation. Comparing the efforts of the USB DMAC instantiation with previous verification of memory isolation of a NIC DMAC [33], strongly suggests that the framework can significantly reduce the cost of verification of isolation (i.e., proving that the invariant is preserved).

Our verification can be extended in two directions. Towards software, the proof obligations can be used to check that device drivers securely configure DMACs or to synthesize security monitors, and the abstract model can be used to check functional correctness (e.g., transmission of network packets). Towards hardware, the model M_3 can be used to either show that a formal hardware design respect the specification, or for model driven testing of closed source hardware.

We plan to implement and model a monitor that runs underneath the Linux USB DMAC driver for the USB DMAC on BeagleBone Black [7], [32], checking that the driver reconfigurations are secure; and then verify that the monitor satisfies the proof obligations. This fulfills two goals: The monitor preserves security even if the driver is buggy, and the monitor itself can be used to detect if the Linux driver has memory isolation bugs.

References

[1] Altera Corporation, “Increase System Performance & Efficiency Using Distributed Direct Memory,” 2004, p. 10. Accessed: April 14, 2022 [Online]. Available: http://xilinx.info/_exhibit/2004/altera/5_SOPC_04_DMA_RF_2_50min.pdf

[2] J. Mangino, "Using DMA with High Performance Peripherals to Maximize System Performance," Texas Instruments Corporation, 2007, p. 13. Accessed: April 14, 2022. [Online]. Available: https://www.ti.com/lit/wp/spna105/spna105.pdf
[3] F. Khunjush and N. J. Dimopoulos, "Extended Characterization of DMA Transfers on the Cell BE Processor," IEEE International Symposium on Parallel and Distributed Processing, 2008, Table 5.
[4] K. Saether, "Using Event Systems and DMA to Cut Power Consumption," techbriefs.com, Table 2. Accessed: April 14, 2022. [Online]. Available: https://www.techbriefs.com/component/content/article/tb/supplements/et/features/articles/6272
[5] T. Enami, K. Kawakami, and H. Yamazaki, "DMA-driven control method for low power sensor node," IEEE Topical Conference on Wireless Sensors and Sensor Networks, 2015.
[6] W.-P. de Roever, et al., "Concurrency Verification: Introduction to Compositional and Non-Compositional Methods," USA: Cambridge University Press, 2012.
[7] BeagleBone Black System Reference Manual. Accessed: April 26, 2022. [Online]. Available: https://github.com/beagleboard/beaglebone-black/wiki/System-Reference-Manual#beaglebone-black-high-level-specification
[8] Z. D. Dittia, G. M. Parulkar, and J. R. Cox Jr, "The APIC Approach to High Performance Network Interface Design: Protected DMA and Other Techniques," Proceedings of the sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies, April 1997.
[9] J. Danisevskis, M. Piekarska, and J.-P. Seifert, "Dark Side of the Shader: Mobile GPU-Aided Malware Delivery," Information Security and Cryptology, pp. 483-495, 2013.
[10] A. Markuze, A. Morrison, and D. Tsafrir, "True IOMMU Protection from DMA Attacks: When Copy Is Faster Than Zero Copy," Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 249-262, March 2016.
[11] F. Zhang, H. Wang, K. Leach, and A. Stavrou, "A Framework to Secure Peripherals at Runtime," Computer Security - ESORICS, pp. 219-238, 2014.
[12] O. Ruwase, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry, "Guardrail: a high fidelity approach to protecting hardware devices from buggy drivers," ACM SIGARCH Computer Architecture News, vol. 42, no. 1, pp. 655-670, March 2014.
[13] P. Stewin, "A Primitive for Revealing Stealthy Peripheral-Based Attacks on the Computing Platform's Main Memory," Research in Attacks, Intrusions, and Defenses, pp. 1-20, 2013.
[14] L. Duflot, Y.-A. Perez, and B. Morin, "What If You Can't Trust Your Network Card?," Recent Advances in Intrusion Detection, pp. 378-397, 2011.
[15] AMD Corporation, "AMD I/O Virtualization Technology (IOMMU) Specification," 2021. Accessed: April 14, 2022 [Online]. Available: https://www.amd.com/system/files/TechDocs/48882_IOMMU.pdf
[16] ARM Limited, "ARM System Memory Management Unit Architecture Specification SMMU architecture version 2.0," 2016. Accessed: April 14, 2022 [Online]. Available: https://documentation-service.arm.com/static/5f900d34f86e16515cdc08fb
[17] J. Yao, V. J. Zimmer, and S. Zeng, "A Tour Beyond BIOS: Using IOMMU for DMA Protection in UEFI Firmware," Intel Corporation, 2017. Accessed: April 14, 2022 [Online]. Available: https://www.intel.com/content/dam/develop/external/us/en/documents/intel-whitepaper-using-iommu-for-dma-protection-in-uefi-820238.pdf
[18] M. Ben-Yehuda et al., "The Price of Safety: Evaluating IOMMU Performance," The 2007 Ottawa Linux Symposium, 2007, p. 9, p. 13.
[19] N. Amit, M. Ben-Yehuda, and B.-A. Yassour, "IOMMU: Strategies for Mitigating the IOTLB Bottleneck," Proceedings of the 2010 international conference on Computer Architecture, pp. 256-274, June 2010, Figure 2.
[20] L. Lei, K. Cong, Z. Yang, and F. Xie, "Validating Direct Memory Access Interfaces with Conformance Checking," Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design, pp. 9-16, Nov. 2014.
[21] A. A. Vasilyev, "Static verification for memory safety of Linux kernel drivers," Proceedings of ISP RAS, vol. 30, no. 6, pp. 143-160, 2018.
[22] J.-J. Bai, T. Li, K. Lu, and S.-M. Hu, "Static Detection of Unsafe DMA Accesses in Device Drivers," 30th USENIX Security Symposium (USENIX Security 21), 2021.
[23] C. Baumann, B. Beckert, H. Blasum, and T. Bormer, "Formal Verification of a Microkernel Used in Dependable Software Systems," Computer Safety, Reliability, and Security, pp. 187-200, 2009.
[24] M. Dam, R. Guanciale, N. Khakpour, H. Nemati, and O. Schwarz, "Formal verification of information flow security for a simple arm-based separation kernel," Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, pp. 223-234, Nov. 2013.
[25] A. Vasudevan et al., "Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework," IEEE Symposium on Security and Privacy, May 2013.
[26] G. Klein et al., "Comprehensive formal verification of an OS microkernel," ACM Transactions on Computer Systems, vol. 32, no. 1, pp. 1-70, Feb. 2014.
[27] R. Gu et al., "Deep specifications and certified abstraction layers," Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 595-608, Jan. 2015.
[28] C. Baumann, M. Näslund, C. Gehrmann, O. Schwarz, and H. Thorsen, "A high assurance virtualization platform for ARMv8," European Conference on Networks and Communications, 2016.
[29] S.-W. Li, X. Li, R. Gu, J. Nieh, and J. Z. Hui, "A Secure and Formally Verified Linux KVM Hypervisor," IEEE Symposium on Security and Privacy (SP), May 2021.
[30] S.-W. Li, X. Li, R. Gu, J. Nieh, and J. Z. Hui, "Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor," Proceedings of the 30th USENIX Security Symposium, 2021.
[31] "HOL Interactive Theorem Prover," https://hol-theorem-prover.org(accessedApril15,2022).
[32] Texas Instruments, "AM335x and AMIC110 Sitara Processors Technical Reference Manual," Rev. P. Accessed: April 15, 2022 [Online]. Available: https://www.ti.com/lit/ug/spruh73q/spruh73q.pdf.
[33] J. Haglund and R. Guanciale, "Trustworthy Isolation of DMA Enabled Devices," Proceedings of 15th International Conference on Information Systems Security, Hyderabad, India, pp. 35-55, Dec. 2019.
[34] M. Kim, Y. Choi, Y. Kim, and H. Kim, "Pre-testing Flash Device Driver through Model Checking Techniques," 1st International Conference on Software Testing, Verification, and Validation, 2008.
[35] E. Alkassar and M. Hillebrand, "Formal Functional Verification of Device Drivers," Proceedings of the 2nd international conference on Verified Software: Theories, Tools, Experiments, pp. 225 - 239, Oct. 2008.
[36] E. Alkassar, M. Hillebrand, S. Knapp, R. Rusev, and S. Tverdyshev, "Formal Device and Programming Model for a Serial Interface," Proceedings of the 4th International Verification Workshop, pp. 4-20, Bremen, Germany, 2007.
[37] J. Duan, "Formal Verification of Device Drivers in Embedded Systems," Ph.D. dissertation, Dept. Computing, Univ. Utah, UT, USA, 2013.
[38] W. Penninckx, J. T. Mühlberg, J. Smans, B. Jacobs, and F. Piessens, "Sound Formal Verification of Linux's USB BP Keyboard Driver," NASA Formal Methods, 2012.
[39] N. Dong, R. Guanciale, and M. Dam, "Refinement-Based Verification of Device-to-Device Information Flow," Formal Methods in Computer Aided Design, 2021.
[40] E.M. Clarke, S. Bose, M.C. Browne, and O. Grumberg, "The Design and Verification of Finite State Hardware Controllers," Technical Report CMU - CS - 87-145 , Carnegie Mellon Univ., July 1987.
[41] H. Hiraishi, K. Hamaguchi, H. Fujii, and S. Yajima, "Regular Temporal Logic Expressively Equivalent to Finite Automata and Its application to Logic Design Verification," Journal of Information Processing, vol. 15, no. 1, pp. 130-138, 1992.
[42] https://github.com/kth-step/dma-controller-verification.git
[43] D. Monniaux, "Verification of Device Drivers and Intelligent Controllers: a Case Study," Proceedings of the 7th ACM & IEEE international conference on Embedded software, pp. 30-36, Sept. 2007.
[44] A. Mera, Y. H. Chen, R. Sun, E. Kirda, and L. Lu "D-Box: DMA-enabled Compartmentalization for Embedded Applications," Network and Distributed Systems Security Symposium, San Diego, CA, USA, April 2022.
[45] J. Haglund and R. Guanciale, "Trustworthy isolation of DMA devices," Journal of Banking and Financial Technology, pp. 75-94, 2020.
[46] A. F. Donaldson, D. Kroening, and P. Rümmer, "Automatic analysis of DMA races using model checking and k-induction," Formal methods in system design, vol. 39, no. 1, pp. 83-113, 2011.
[47] O. Schwarz and C. Gehrmann, "Securing DMA through virtualization," Proceedings of Complexity in Engineering, 2012.

[48] O. Schwarz and M. Dam, “Formal Verification of Secure User Mode Device Execution with DMA,” Hardware and Software: Verification and Testing, pp. 236-251, Haifa, Isreal, 2014.
[49] M. Yu, V. Gligor, and L. Jia, “An I/O Separation Model for Formal Verification of Kernel Implementations,” IEEE Symposium on Security and Privacy, San Francisco, CA, USA, May 2021.
[50] L. Ryzhyk, P. Chubb P, I. Kuz, E. Le Sueur, and G. Heiser, “Automatic device driver synthesis with termite,” Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, pp. 73-86, 2009.
[51] L. Ryzhyk et al., “User-guided device driver synthesis,” Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation, pp. 661-676, 2014.
[52] J. Sawada and W. A. Hunt Jr., “Verification of FM9801: An Out-of-Order Microprocessor Model with Speculative Execution, Exceptions, and Program-Modifying Capability,” Formal Methods in System Design vol. 20, pp. 187-222, 2002.
[53] M. N. Velev and P. Gao, “Automatic formal verification of multithreaded pipelined microprocessors,” IEEE/ACM International Conference on Computer-Aided Design, 2011.
[54] P.-M. Seidel, “Formal Verification of an Iterative Low-Power x86 Floating-Point Multiplier with Redundant Feedback”, 10th International Workshop on the ACL2 Theorem Prover and its Applications, pp. 70-83, 2011.
[55] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneide, “Device Driver Safety Through a Reference Validation Mechanism,” Proceedings of the 8th USENIX conference on Operating systems design and implementation, Dec. pp. 241-254, 2008.
[56] M. Grisafi, M. Ammar, M. Roveri, and B. Crispo, “PISTIS: Trusted Computing Architecture for Low-end Embedded Systems,” 31st USENIX Security Symposium, 2022.