Memory Security in Reconfigurable Computers: Combining Formal Verification with Monitoring

Tobias Wiersema and Stephanie Drzevitzky and Marco Platzner
University of Paderborn, Germany
Email: tobias.wiersema@uni-paderborn.de, stephanie@drzevitzky.de, platzner@uni-paderborn.de

***Abstract*—Ensuring memory access security is a challenge for reconfigurable systems with multiple cores. Previous work introduced access monitors attached to the memory subsystem to ensure that the cores adhere to pre-defined protocols when accessing memory. In this paper, we combine access monitors with a formal runtime verification technique known as proof-carrying hardware to guarantee memory security. We extend previous work on proof-carrying hardware by covering sequential circuits and demonstrate our approach with a prototype leveraging ReconOS/Zynq with an embedded ZUMA virtual FPGA overlay. Experiments show the feasibility of the approach and the capabilities of the prototype, which constitutes the first realization of proof-carrying hardware on real FPGAs. The area overheads for the virtual FPGA are measured as 2x-10x, depending on the resource type. The delay overhead is substantial with almost 100x, but this is an extremely pessimistic estimate that will be lowered once accurate timing analysis for FPGA overlays become available. Finally, reconfiguration time for the virtual FPGA is about one order of magnitude lower than for the native Zynq fabric.**

I. Introduction

The density of todays reconfigurable hardware devices allows for implementing single-chip reconfigurable computers with a large number of modules or cores. Through dynamic or even partial reconfiguration modules can be loaded on demand, increasing flexibility. In such multi-core systems, memory access security becomes a major concern. Several modules that access the same physical memory need to adhere to a specified policy governing their access patterns. A simple policy, for example, is to enforce that each core can only access its own segment of the memory. There are, however, more involved policies in use when it comes to intended sharing of data between cores, to the handling of conflict-of-interest classes, or to different security levels [1].

Previously, Huffmire et al. [2] introduced a monitoring-based approach to ensure memory access security in reconfigurable computers. They presented a formal language and a compilation tool flow that allows a designer to specify a memory access policy and generate a circuit for a so-called memory access monitor. All modules' memory accesses have to be routed through the monitor, enabling the monitor to block any memory access that violates the policy at runtime. Our work takes up the monitoring concept but strives for guaranteeing memory access security in the strength of formal verification. To that end, we leverage the recently presented concept of proof-carrying hardware [3]. Proof-carrying hardware requires the producer of a circuit to generate a formal proof of a desired property of that circuit and transmit the circuit implementation in form of an FPGA bitstream together with the proof to a consumer. The consumer verifies the proof and, if successful, loads and uses the circuit. Since generating a proof from scratch typically is more resource-consuming than verifying it, proof-carrying hardware is well-suited for consumers that cannot afford the long runtimes and computing resources for full formal verification. In particular, consumers that are required to receive and configure new modules during runtime can benefit from the proof-carrying hardware approach.

The main contribution of this paper is a novel solution for memory access security by bringing together the monitoring approach with the proof-carrying hardware concept. The consumer operates a reconfigurable resource where several cores access shared memory. All memory accesses are routed through a memory access monitor that implements a pre-defined memory access policy. The policy can change during runtime to reflect different applications and security requirements. The consumer receives new monitors together with a proof of their functional correctness, verifies the proof, and in case of success partially reconfigures the monitor.

A further novel contribution of this paper is the prototypical implementation on a Xilinx Zynq platform running the ReconOS [4] operating system. To realize our prototype we extend related work on proof-carrying hardware in two directions. First, we show how to verify sequential circuits that are required for implementing more complex, dynamic memory access policies. Related work focused on combinational circuits. Second, we utilize ZUMA [5] as a recent FPGA overlay to be able to access the bitstream details for verification. Bitstreams for the overlay are generated using the VTR [6] tool flow. We embed the overlay into the Zynq/ReconOS system which enables us to use a mature infrastructure for implementing hardware/software systems, including a CPU core, memory controller, peripherals and a standard software operating system. In contrast, related work relied on an abstract FPGA architecture that could only be simulated.

The remainder of the paper is structured as follows: Section II provides a review of related work, mainly in memory access monitors and proof-carrying hardware. Section III discusses our concept for combining runtime verification with memory access monitors. We then present our prototype system and tool flow as well as experimental results in Section IV, and conclude the paper with Section V.

978-1-4799-6245-7/14/$31.00 ©2014 IEEE

II. Related work

This section discusses related work in the two main areas relevant for our work, memory security and reference monitors for reconfigurable computers and proof-carrying hardware for runtime verification of reconfigurable modules.

A. Memory Security and Reference Monitors

Memory security in reconfigurable computers has been an active field in the past years. However, many of the presented projects focus on memory security in multi-core processors and use FPGA technology for prototyping. For example, Crenne et al. [7] discuss data integrity and confidentiality and propose a special security core for protection of application loading and secure execution. Eckert et al. [8] present a malware scanner and filter for DMA-copied data implemented by a watchdog module. The malware scanner can be adapted by partial reconfiguration. Their prototype architecture uses an ARM running Linux and places the watchdog between an arbiter and a memory controller, which is close to our experimental setup. Basile et al. [9] considered the execution of code in an untrusted environment and used an FPGA within this environment as a core of trust. The core of trust relies on hardware monitors verifying the integrity of the transmitted code before and during execution. Cotret et al. [10] looked at the bus system in a multi-core and proposed to add watchdogs or fire-walls to cores and memory in a distributed fashion to protect them. A different line of research focuses on the security of the FPGA configuration. In a recent survey, Durvaux et al. [11] gave an overview over methods to prevent recovering bitstreams and thus cloning intellectual property and to ensure adherence to the licensing terms of license, while retaining the FPGA-typical flexibility.

In the following we discuss in more detail the memory reference monitors introduced by Huffmire et al. [2], [12], [13]. In contrast to other related work, they allow for the specification and enforcement of arbitrary memory access policies between a number of cores, i.e., CPU cores or hardware modules. The memory reference monitor is the only module in the system that has direct memory access. All other cores have to route their memory accesses via the memory reference monitor. According to a pre-defined memory access policy between the cores, the monitor either grants or denies memory accesses. In order to describe memory access policies, Huffmire et al. designed a formal language. A memory access policy is defined by the memory accesses it allows, and each of the accesses is defined by a module, a range, and an access type. Modules denote the cores mapped to the FPGA that request memory access of a certain access type, e.g., read, write, scrub. Ranges are segments of the memory.

As an example, a policy for the static isolation of two modules, where Module_1 has complete access to only Range_1, and Module_2 has complete access to only Range_2, is expressed with the following policy grammar:

$$
\begin{array}{lcl}
\text{rw} & \longrightarrow & \text{r} \mid \text{w}; \\
\text{Range}_1 & \longrightarrow & [0\text{x}8\text{e}7\text{b}008, 0\text{x}8\text{e}7\text{b}00\text{f}]; \\
\text{Range}_2 & \longrightarrow & [0\text{x}8\text{e}7\text{b}018, 0\text{x}8\text{e}7\text{b}01\text{b}]; \\
\text{Access} & \longrightarrow & \{\text{Module}_1, \text{rw}, \text{Range}_1\} \mid \\
 & & \{\text{Module}_2, \text{rw}, \text{Range}_2\}; \\
\text{Policy} & \longrightarrow & (\text{Access})^*;
\end{array}
$$

Besides static policies, the formal language developed by Huffmire et al. also allows designers to describe more complex dynamic policies. An example for a dynamic policy is the *chinese wall*, which encodes so-called conflict-of-interest classes. Assume a reconfigurable system with an AES crypto core and two other modules that need crypto services, but must not access each other's data. While in general the AES core needs access to the memory areas of both modules, a designer might wish to specify that while the AES core encrypts or decrypts data for one of the modules, it must not be allowed to access data of the other module. Hence, the two modules belong to one conflict-of-interest class.

Consider a system with five modules, Module_1 to Module_4 and Module_{AES}, and corresponding memory ranges Range_1 to Range_4. Further, consider two conflict-of-interest classes, one with Module_1 and Module_2 and another one with Module_3 and Module_4. Omitting the range and *rw* definitions for simplicity, the policy the AES core has to adhere to during one encryption/decryption task is formulated as follows:

$$
\begin{array}{lcl}
\text{Access}_1 & \longrightarrow & \{\text{Module}_{AES}, \text{rw}, (\text{Range}_1 \mid \text{Range}_3)\}^*; \\
\text{Access}_2 & \longrightarrow & \{\text{Module}_{AES}, \text{rw}, (\text{Range}_1 \mid \text{Range}_4)\}^*; \\
\text{Access}_3 & \longrightarrow & \{\text{Module}_{AES}, \text{rw}, (\text{Range}_2 \mid \text{Range}_3)\}^*; \\
\text{Access}_4 & \longrightarrow & \{\text{Module}_{AES}, \text{rw}, (\text{Range}_2 \mid \text{Range}_4)\}^*; \\
\text{Policy} & \longrightarrow & \text{Access}_1 \mid \text{Access}_2 \mid \text{Access}_3 \mid \text{Access}_4;
\end{array}
$$

If the first access of Module_{AES} is to Range_3, any subsequent access to Range_4 will be blocked by the monitor. Accesses to Range_3 as well as Range_1 and Range_2 are valid, resulting in Access_1 and Access_3, respectively.

In addition to the formal language, Huffmire et al. also presented a method for synthesizing a policy into a monitor circuit in hardware description language Verilog [2]. To this end, they build a syntax tree from the policy, expand it into a regular expression, convert that expression into a non-deterministic finite automaton, and construct a minimized state machine from it. They implemented prototypes for several example policies, differing in complexity of the policy, number of modules, and number of memory ranges. Huffmire et al. extended their monitor-and-enforce approach for memory accesses to ensuring that modules do not unintentionally or maliciously communicate. They propose the concepts of moats and drawbridges on the level of physical routing to isolate all modules from each other and allow only necessary connections between them [12].

B. Proof-Carrying Hardware

The concept of proof-carrying hardware (PCH) was proposed by Drzevitzky et al. [3]. PCH is the reconfigurable hardware equivalent of proof-carrying code, an approach introduced earlier by Necula and Lee [14]. The PCH concept distinguishes a circuit producer, e.g., a design center, and a consumer, e.g., a data center operating a reconfigurable computer. The consumer wants to load and execute a reconfigurable hardware module that was created by the producer. Additionally, the consumer specifies a security property that the module needs to fulfill and, before loading, requires a formal proof of the property. It is the task of the producer to generate not only the module but also the proof and transmit both to the consumer. In PCH, the module implementation and the proof have been denoted as proof-carrying bitstream. The consumer will then verify that the proof is correct and that the proof actually belongs to the module. Since checking a proof is typically much easier than creating it, the effort in runtime and resources for formal verification is with the producer. PCH is thus well-suited for consumers with low resources or for consumers that use dynamic reconfiguration and thus can not invest in the substantial runtime needed for full formal verification.

PCH is a rather general concept and can be used with different functional or non-functional security properties. PCH is also a powerful concept since it does not require any previously established trust in the producer, his tools, or the communication channels. The approach is even robust against third parties performing man-in-the-middle attacks, even when circumstances require the security policy to be public. Since the consumer always knows the original security policy and uses it to verify the proof, the security check would fail if a modified policy was used to create a false proof, or if either section of the transmitted proof-carrying bitstream, i.e., the hardware module itself or the proof, had been intentionally tampered with or accidentally damaged. If the checks on the consumer side succeed, it is guaranteed that i) the proof matches the code and the consumer's security policy and ii) the code has the proven property. What is needed is a minimal set of trusted tools at the consumer side (see III-C).

In their work, Drzevitzky et al. defined the trust and threat model for proof-carrying hardware and applied the approach to runtime checking of combinational equivalence as rather broad security property. They modeled the combinational equivalence of a circuit implementation with its specification using a reduction to a satisfiability (SAT) solving test, and proposed to use the resolution trace of the SAT solver as proof. The authors also implemented a prototypical tool flow for this runtime check based on the open source hardware synthesis tool flow VTR [6]. Since this tool flow cannot produce bitstreams for commercially available FPGAs, the work of Drzevitzky et al. was limited to abstract FPGA architectures as targets. They experimented with adders and multipliers of varying complexities and showed that the producer actually bore the majority of the workload for runtime combinational equivalence checking, and that the consumer was left with a comparatively easy problem for the validation of the proof. Multipliers with large bit-widths led to excessive proof generation times, which is a scaling behavior known from static verification techniques as well. Experiments with benchmarks from the SAT race 2008 confirmed that the workload shift towards the producer is also noticeable for more complex circuits.

III. Design and Runtime Verification Flow

In this section we describe the overall design flow depicted in Figure 1, that splits into a design time and runtime part executed by the consumer and the producer of the memory access monitor circuit. The description concentrates on the creation and runtime verification of the monitor circuits. For the creation of the hardware modules or cores themselves, non-PCH tool flows are sufficient.

A. Consumer: Memory Access Policy Specification

The consumer specifies the desired functionality of the memory access monitor and sends this information to the producer. For specifying the memory access policy we use behavioral Verilog. Alternatively, a description in the formal language developed by Huffmire et al. could also be used. The requirement on the specification mechanism is that the producer must be able to synthesize an implementation from it, and the consumer must be able to derive a miter function from the specification. As security property we employ functional equivalence, i.e., the consumer will receive a proof that guarantees that the specification and the implementation received from the producer are combinationally or sequentially equivalent, respectively.

The consumer-producer scenario we display in Figure 1 is the simplest possible. Without loss of generality, a third party could specify the memory policies and security properties and order an implementation from a producer. The consumer would then receive the specification from the third party and the resulting proof-carrying bitstream from the producer and proceed as described in the following.

B. Producer: Implementation and Proof

The producer receives the design specification and synthesizes it into an FPGA bitstream, using the tool flow of Huffmire et al. and, subsequently VTR for Verilog synthesis and place & route. After that, the producer re-extracts the logic function from the bitstream and, together with the original design specification, computes the miter function. The miter function is shown in Figure 2 and is constructed such that the output of the miter, i.e., the error flag, can only be 1 if the specification and implementation differ for at least one input vector. For proving functional equivalence for combinational circuits, it is thus sufficient to prove unsatisfiability of the miter. We use ABC [15] to construct a miter in conjunctive normal form and the SAT solver PicoSAT [16] to prove unsatisfiability. PicoSAT also generates a proof trace which, together with the bitstream, is send to the consumer.

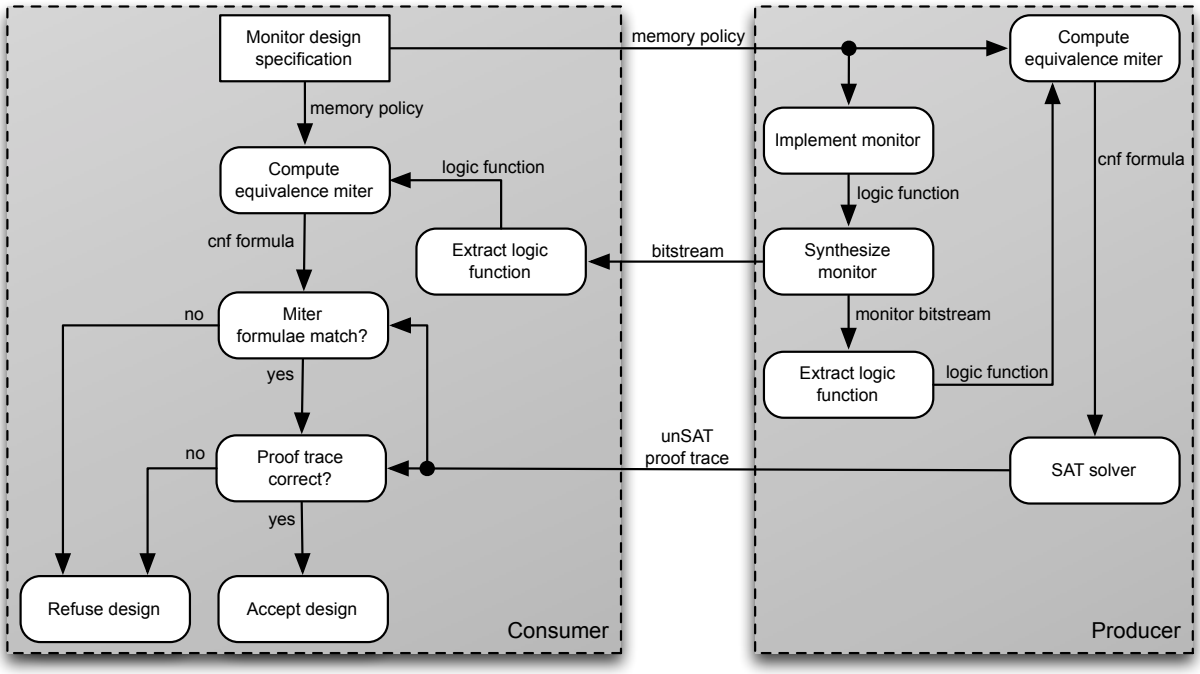

Fig. 1. Complete design flow overview. The consumer starts by sending the design and security specifications to the producer, and ends with either accepting or refusing the design.

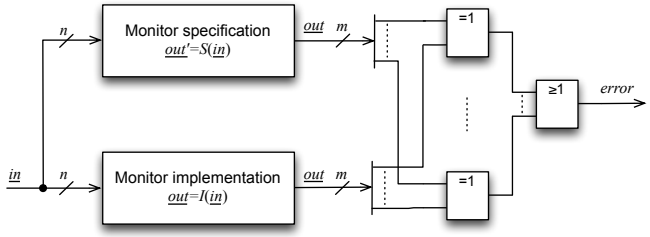

Fig. 2. Miter $M(S(x), I(x))$ for proving the functional equivalence of specification S and implementation I, (cp. [3]).

Dynamic memory access policies lead to sequential monitor circuits. While our tool flow for verifying combinational circuits conceptually follows that from related work [3], we have extended the concept and tool flow to also work with sequential miters using bounded sequential equivalence checking. A sequential miter circuit is unrolled for a specified number n of time frames, Figure 3 contains an example with $n = 3$. The resulting circuit, unrolled for n time frames, contains n copies of the circuit, connected at its flip flops. Every time frame thus represents one clock cycle, and we can change the primary inputs, and observe the primary outputs at every individual cycle. The miter construction then compares all outputs in each time frame and the flip flop signals of the last frame, and raises the error flag if there is a deviation somewhere. As we have to choose a specific amount of unrolling time frames, we observe that the compiled monitors are essentially state machines, and their internal transitions only depend on their current state and the new input. Suppose there is an input sequence i which satisfies the miter function, i.e., it leads to different outputs for the implemented circuit and the specification, and the corresponding state transition path of the state machine contains cycles. Then the input sequence i', which leaves out all state cycles of i, is also a valid input sequence and it also satisfies the miter. If the miter is thus provably unsatisfiable for all maximum length cycle free state transition paths, it is unsatisfiable for all input sequences of all lengths. Hence we can simply use a number n of unrolling frames larger than the number s of automaton states, to ensure that every cycle free sequence has been considered.

C. Consumer: Runtime Verification

The consumer receives the bitstream for the monitor circuit together with the proof trace for unsatisfiability. In a first step, the consumer also extracts the monitor's logic function from the bitstream and forms a miter in conjunctive normal form in the same way as the producer, but with the original specification. The so-created miter is compared to the miter sent by the producer, which is part of the proof trace. If the miters do not match, then the proof is not based on the desired functionality and the monitor is refused. If the miters match the consumer verifies the proof by checking each reduction step in the proof trace until an empty clause results. Only then, the implementation is shown to adhere to the security property and the monitor is accepted.

In terms of computational complexity, the consumer steps in the tool flow are simpler than the producer steps. In particular, proving unsatisfiability is much harder than checking a proof trace. It has to be mentioned, however, that in the worst case

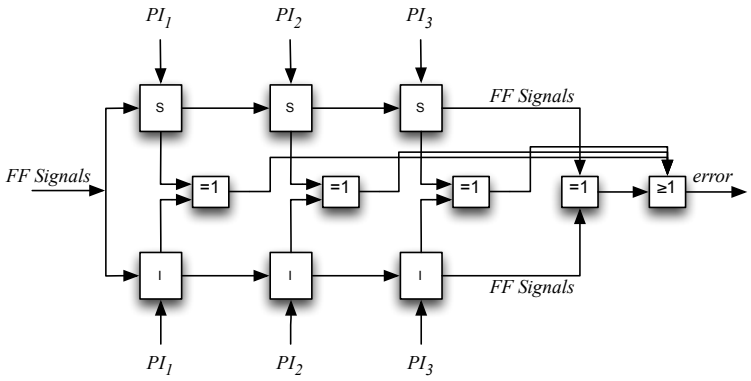

Fig. 3. Time bounded sequential miter $M_3(S(x), I(x))$ for proving the functional equivalence of specification S and implementation I for three cycles.

the size of the proof trace can be exponential in number of variables of the miter conjunctive normal form. Practically, the effort of the consumer depends strongly on the effort to re-extract the monitor function from the bitstream. In our prototype implementation we thus leverage a virtual FPGA architecture and the open source hardware synthesis tool flow VTR, so that the syntax and semantics of the resulting bitstream is known.

The only tools the consumer has to trust here are the one extracting the logic function and the tool which checks the proof trace. It is therefore also beneficial from a security and trust viewpoint, to favor solutions where the logic function extraction is easy. For the proof trace checker we can benefit from the popularity of the SAT-based hardware verification approach, which has lead to a host of tools and standardized file formats, so that the consumer can compare the outputs of several mature tools, before trusting the result.

IV. Evaluation

To demonstrate the capability of our proposed approach for ensuring memory security, we have built a prototypical system. As platform we chose a ZedBoard containing a Xilinx Zynq-7000 system-on-a-chip with a dual ARM Cortex-A9 MPCore, and 512 MB RAM. In the following we first describe our prototype and then present experimental results.

A. Prototype Architecture

Our prototype architecture embeds a virtual FPGA overlay into a reconfigurable system as shown in Figure 4. We use a virtual FPGA since we need to be able to interpret the transmitted configuration bitstream for the memory access monitor. FPGA vendors typically do not share the necessary information, and reverse engineering the bitstream or additionally transmitting and interpreting low-level circuit descriptions such as Xilinx XDL are extremely tedious processes. Virtual FPGAs or FPGA overlay architectures have become increasingly popular in the last years for a number of reasons. They provide a means to implement portable circuits, to bring partial reconfiguration capabilities to FPGAs that have no native support of this feature, to achieve fast configuration rates, to prototype coarse grained arrays, or to be able to implement circuits created with open source tool flows such as VTR [6] on real FPGAs.

In our work, we leverage the virtual FPGA overlay ZUMA [5]. We have chosen ZUMA as it is open source, readily programmable using the open source tool flow VTR, uses area-conserving techniques to reduce the overhead and is already available in implementations for Xilinx and Altera FPGAs. While a detailed presentation of ZUMA with its capabilities and overheads is beyond the scope of this paper, we outline its main features: ZUMA defines an island style virtual FPGA architecture with configuration options for the number of logic blocks, lookup tables per block, connectivity between and inside the blocks, as well as track width and wire length. The ZUMA switch boxes are not fully connected for a reasonable trade-off FPGA between area and routability. To reduce the area overhead for implementing the overlay, ZUMA implements its virtual lookup-tables via LUTRAMs in modern FPGAs [5] or, as proposed more recently, uses physical wires as virtual routing resources through runtime reconfiguration [17]. We have extended the basic ZUMA reference implementation with registers to be able to map sequential circuits to the virtual FPGA overlay.

We embed the ZUMA overlay into ReconOS [4], [18]. ReconOS is an execution environment for hybrid hardware/software systems featuring a multithreaded programming model which allows for regular software threads as well as hardware threads. Hardware threads are basically circuits that can interact with other threads and system resources, such as semaphores, through an operating system interface. To enable this interaction, ReconOS instantiates one delegate thread in software per running hardware thread. The delegate thread accesses the operating systems services and communicates with other threads on behalf of the hardware thread. The use of ReconOS enables us to use a mature, Linux-based infrastructure for implementing hardware/software systems, including a CPU core, memory controller, peripherals and a standard software operating system.

As shown in Figure 4, we have modified the ReconOS arbiter in the memory access path of the hardware threads to

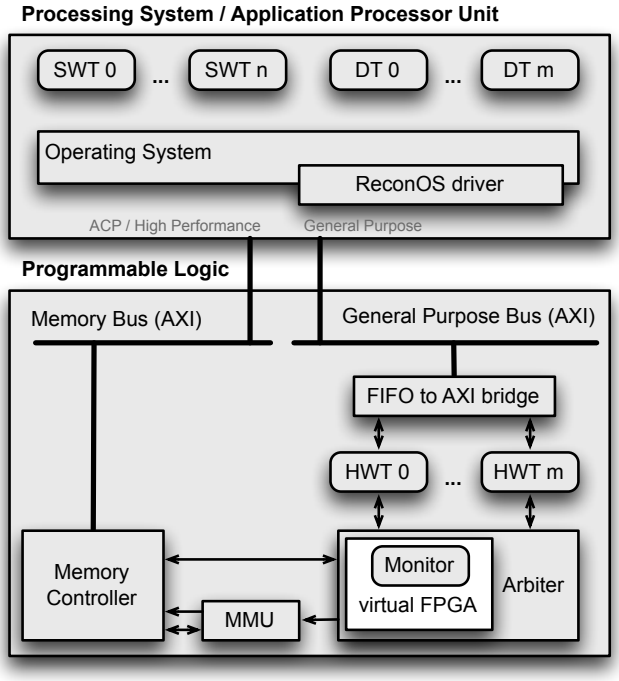

Fig. 4. Xilinx Zynq version of ReconOS [4], with n software threads (SWT), m hardware threads (HWT), their m delegate threads (DT), and an arbiter including a memory monitor in the memory access path of the HWTs.

include a memory access monitor. The access monitor itself is implemented in our ZUMA virtual FPGA overlay. The arbiter provides to the monitor as inputs the virtual memory address, the type of the request (read or write) and its source, the hardware thread identifier. In fact, this implementation is in line with the first alternative described by Huffmire et al. in [2, p. 207, Fig. 9].

B. Experimental Results

We have conducted a series of experiments to investigate different aspects of our approach and prototype. First, we have evaluated the feasibility of the proof-carrying hardware approach by computing the necessary proofs for a variety of example policies in different complexities. Table I presents the runtimes for the consumer and producer for different memory access policies taken from [2] and [19]. The Biba (biba) as well as the Low Watermark (low) models implement data integrity, the Bell and LaPadula (bl) as well as the High Watermark (high) models realize data confidentiality, the Isolation (iso) model gives a simple separation of memory ranges for data isolation and, finally, the Chinese Wall (chin) model enforces conflict-of-interest classes. The different versions of each policy are variations with different number of ranges, modules or conflict-of-interest classes.

The runtimes listed for both consumer and producer are the sum of the runtimes for the necessary steps on the respective side, as depicted in Figure 1. The consumer runtime thus includes as main parts the computation of the miter function and the check of the resolution proof trace, while the producer

	Consumer	Producer		Proof size
Policy	total [s]	total [s]	workload	
biba1	0.141	1.043	88.09 %	10.89 KiB
biba2	0.132	1.100	89.29 %	11.64 KiB
biba3	0.141	1.035	88.01 %	20.46 KiB
biba4	0.135	1.002	88.13 %	8.52 KiB
biba5	0.139	1.039	88.20 %	15.50 KiB
biba6	0.136	1.077	88.79 %	25.92 KiB
bl1	0.132	1.015	88.49 %	10.82 KiB
bl2	0.133	1.031	88.57 %	11.14 KiB
bl3	0.131	1.069	89.08 %	18.99 KiB
bl4	0.130	1.008	88.58 %	8.52 KiB
bl5	0.136	1.006	88.09 %	15.04 KiB
bl6	0.134	1.076	88.93 %	24.29 KiB
iso1	0.130	1.004	88.54 %	7.62 KiB
iso2	0.131	1.035	88.77 %	13.89 KiB
iso3	0.164	1.567	90.53 %	100.13 KiB
iso4	0.195	1.700	89.71 %	186.53 KiB
high1	1.213	2.483	67.18 %	.43 KiB
high2	1.426	2.872	66.82 %	.49 KiB
high3	3.814	9.509	71.37 %	3.73 KiB
high4	1.216	2.476	67.06 %	.50 KiB
high5	3.092	11.865	79.33 %	3.38 KiB
high6	4.319	11.018	71.84 %	3.48 KiB
low1	1.270	2.600	67.18 %	.43 KiB
low2	1.386	2.829	67.12 %	.43 KiB
low3	3.895	9.389	70.68 %	2.32 KiB
low4	1.243	2.560	67.32 %	.43 KiB
low5	2.432	7.220	74.80 %	.69 KiB
low6	4.350	10.894	71.46 %	2.96 KiB
chin1	2.659	6.645	71.42 %	2.04 KiB
chin2	4.203	6.812	61.84 %	1.94 KiB

TABLE I
RUNTIME COMPARISON BETWEEN CONSUMER AND PRODUCER.

Prototype version	MMU	Monitor	Memory policy
RV_{ref}	Yes	None	None
$RV_{arb-Zynq}$	Yes	Zynq	Fixed
$RV_{arb-Zuma}$	Yes	ZUMA	Exchangeable

TABLE II
PROTOTYPE VERSIONS FOR OUR EXPERIMENTS.

runtime mainly consist of the synthesis of the monitor to the virtual FPGA overlay, the computation of the miter function and the computation of the unsatisfiability proof. Table I clearly shows that the producer bears the computational burden of establishing the consumer's trust in the module. If we interpret the sum of the producer and consumer runtimes as the cost of trust, then the column "workload" in Table I lists the producer's share of this cost. According to the claim of proof-carrying hardware this share should be significantly larger than 50%, which is the case in our experiment with the lowest share being 61.84% for policy chin2 and the highest share being 90.53% for policy iso3. Table I further lists the sizes of the resolution proofs which constitute the variable part of the proof-carrying bitstreams. The ZUMA bitstreams themselves were 4.7KiB large. Since we do not use bitstream compression, the ZUMA bitstream size only depends on the dimension and architectural configuration of the overlay, and not on the implemented circuits.

Performance measure	RV_{ref}	$RV_{arb-Zynq}$	$RV_{arb-Zuma}$
write access [cycles]	18	18	18
read access [cycles]	37	41	41
area [LUTs / fraction]		4,570 / 8% – 4,722 / 8%	9661 / 18 %
area [LUTRAMs / fraction]		517 / 2% – 517 / 2%	5393 / 30 %
max frequency [MHz]		86.36 – 109.89	1.1

TABLE III
PERFORMANCE RESULTS OF THE THREE RECONOS VERSIONS.

In a second series of experiments, we studied the ReconOS/Zynq prototype implementation and its overheads. We compared the three different versions listed in Table II. RV_{ref} is a ReconOS reference version using direct memory access with virtual memory support, but no memory access monitor. $RV_{arb-Zynq}$ is a version where we have synthesized monitor circuits for different policies directly to the Zynq reconfigurable fabric and included them into the arbiter module. In Table II we denote the memory policy for this version as static, since in our experiments we created a full system configuration for each policy. However, using partial reconfiguration of the Zynq the policies could be made exchangeable. Finally, $RV_{arb-Zuma}$ is the version employing the ZUMA virtual FPGA overlay embedded into the arbiter module. $RV_{arb-Zuma}$ enables us to employ our proof-carrying hardware approach and, in addition, to quickly exchange the memory access policy. We have chosen the size of the overlay just large enough to accommodate the largest of the test circuits of Table I. Thus, $RV_{arb-Zuma}$ uses an overlay with 6×5 clusters, a cluster comprises 4 basic logic elements (BLE), and one BLE contains one 6-input lookup table (LUT) and an optional flip flop (FF). The routing resources are arranged in island style around the clusters and contain 60 wires per track.

Table III shows the performance comparison for the three prototype versions. In terms of clock cycles needed for read and write accesses, the overheads of the versions with memory access monitors are rather small. We have averaged the measurements for the access times over 100 consecutive accesses to level out the effects of ReconOS' translation lookaside buffers and the MMU. In Table III we compare the number of clock cycles that a hardware thread is stalled while waiting for ReconOS' memory controller. Memory write accesses in ReconOS are implemented asynchronously, i.e., the hardware thread does not receive or wait for a write confirmation. Therefore, neither $RV_{arb-Zynq}$ nor $RV_{arb-Zuma}$ adds any delay to writes. For read requests, RV_{ref} needed an average of 37 clock cycles to return the data to the hardware thread, while both $RV_{arb-Zynq}$ and $RV_{arb-Zuma}$ added 4 cycles, two to drain the command pipeline and feed the request to the monitor, one to get its result, and one to repopulate the pipeline. The read access cycles have been measured for single word accesses which is pessimistic since typical hardware threads read data in bursts.

Rows 4-5 of Table III list the area overheads for the arbiter including the access monitors. For $RV_{arb-Zynq}$, we measured the required area on the Zynq for all policies and give the range from lowest to highest area requirement; for $RV_{arb-Zuma}$ the overhead is constant since we have chosen the size of the overlay to match the largest monitor design. As expected, the overlay comes with a high area overhead. Compared to the native Zynq implementation, the overlay more than doubles the number of required LUTs, mainly because the demand for LUTRAMs in $RV_{arb-Zuma}$ is more than 10-fold compared to $RV_{arb-Zynq}$.

The greatest disadvantage for the ZUMA overlay seems to be the clock frequency which, as shown in Table III, reduces to 1.1 MHz. However, this maximum clock frequency corresponds to the longest combinational path in the overlay's circuit as identified by the Xilinx timing analyzer. This is an extremely pessimistic bound on the delay of a circuit since it basically assumes chaining all BLEs together in a long path without including any registers. While in our experiments we operated all modules, including the ZUMA overlays, with 100 MHz, and never experienced an error, a tool for estimating the maximum clock frequency for a specific configuration of the overlay is part of our future work. Also in related work, there is a lack of accurate timing analyses for FPGA overlays.

Finally, we measured the reconfiguration time for our FPGA overlay with 15,296 cycles at 100 MHz or 153μs for a naive, sequential reconfiguration approach. As stated in [5], by using parallel configuration paths this number can be made as low as 2^6 cycles or $640ns$ per configuration in our case. Partial reconfiguration of an area corresponding to the size of the largest monitor in ReconOS/Zynq takes around one ms.

V. CONCLUSION

In this paper we have presented the combination of memory access monitors with formal runtime verification using a proof-carrying hardware approach. We have implemented the first known prototype realizing proof-carrying hardware on real FPGAs by embedding a virtual FPGA overlay into a ReconOS/Zynq system. The prototype demonstrates the feasibility of the concept. The measured performance parameters for the FPGA overlay indicate substantial overheads in area and delay. Future work includes reducing the overheads for the FPGA overlay, for example by a more accurate timing analysis for the overlay circuit that takes the actual overlay configuration into account, and exploring the possibility of verifying non-functional properties of circuits with the proof-carrying hardware approach.

ACKNOWLEDGMENT

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre On-The-Fly Computing (SFB 901).

REFERENCES

[1] F. B. Schneider, "Enforceable security policies," *ACM Trans. Inf. Syst. Secur.*, vol. 3, no. 1, pp. 30–50, Feb. 2000. [Online]. Available: http://doi.acm.org/10.1145/353323.353382

[2] T. Huffmire, T. Sherwood, R. Kastner, and T. Levin, "Enforcing memory policy specifications in reconfigurable hardware," *Computers & Security*, vol. 27, no. 56, pp. 197 – 215, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167404808000138

[3] S. Drzevitzky, U. Kastens, and M. Platzner, "Proof-carrying hardware: Concept and prototype tool flow for online verification," *International Journal of Reconfigurable Computing*, vol. 2010, p. 11, 2010.

[4] A. Agne, M. Happe, A. Keller, E. Lubbers, B. Plattner, M. Platzner, and C. Plessl, "ReconOS: An operating system approach for reconfigurable computing," *IEEE Micro*, vol. 34, no. 1, pp. 60–71. [Online]. Available: http://dx.doi.org/10.1109/MM.2013.110

[5] A. Brant and G. G. Lemieux, "ZUMA: An open FPGA overlay architecture," in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. IEEE, 2012, pp. 93–96.

[6] J. Rose, J. Luu, C. W. Yu, O. Densmore, J. Goeders, A. Somerville, K. B. Kent, P. Jamieson, and J. Anderson, "The VTR project: Architecture and CAD for FPGAs from Verilog To Routing," in *Proc. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2012, pp. 77–86.

[7] J. Crenne, R. Vaslin, G. Gogniat, J.-P. Diguet, R. Tessier, and D. Unnikrishnan, "Configurable memory security in embedded systems," *ACM Transactions on Embedded Computing Systems*, vol. 12, no. 3, pp. 71:1–71:23, Mar 2013. [Online]. Available: http://dx.doi.org/10.1145/2442116.2442121

[8] M. Eckert, I. Podebrad, and B. Klauer, "Hardware based security enhanced direct memory access," in *Communications and Multimedia Security*, ser. Lecture Notes in Computer Science, B. De Decker, J. Dittmann, C. Kraetzer, and C. Vielhauer, Eds. Springer Berlin Heidelberg, 2013, vol. 8099, pp. 145–151. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-40779-6_12

[9] C. Basile, S. Di Carlo, and A. Scionti, "FPGA-based remote-code integrity verification of programs in distributed embedded systems," *IEEE Trans. Syst., Man, Cybern. C*, vol. 42, no. 2, pp. 187–200, 2012. [Online]. Available: http://dx.doi.org/10.1109/TSMCC.2011.2106493

[10] P. Cotret, G. Gogniat, J.-P. Diguet, and J. Crenne, "Lightweight reconfiguration security services for axi-based MPSoCs," *22nd International Conference on Field Programmable Logic and Applications (FPL)*, 2012. [Online]. Available: http://dx.doi.org/10.1109/FPL.2012.6339233

[11] F. Durvaux, S. Kerckhof, F. Regazzoni, and F.-X. Standaert, "A survey of recent results in fpga security and intellectual property protection," in *Secure Smart Embedded Devices, Platforms and Applications*, K. Markantonakis and K. Mayes, Eds. Springer New York, 2014, pp. 201–224. [Online]. Available: http://dx.doi.org/10.1007/978-1-4614-7915-4_9

[12] T. Huffmire, T. Levin, T. Nguyen, C. Irvine, B. Brotherton, G. Wang, T. Sherwood, and R. Kastner, "Security primitives for reconfigurable hardware-based systems," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 3, pp. 10:1–10:35, May 2010.

[13] T. Huffmire, B. Brotherton, N. Callegari, J. Valamehr, J. White, R. Kastner, and T. Sherwood, "Designing secure systems on reconfigurable hardware," *ACM Trans. on Design Automation of Electronic Systems*, vol. 13, no. 3, pp. 1–24, July 2008.

[14] G. Necula and P. Lee, "Proof-carrying code," School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, Tech. Rep. CMU-CS-96-165, November 1996.

[15] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Computer Aided Verification*, T. Touili, B. Cook, and P. Jackson, Eds. Springer, 2010, vol. LNCS 6174, pp. 24–40.

[16] A. Biere, "PicoSAT essentials," *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, vol. 4, pp. 75–97, 2008.

[17] D. Koch, C. Beckhoff, and G. G. Lemieux, "An efficient fpga overlay for portable custom instruction set extensions," in *Proc. IEEE International Conference on Field Programmable Logic and Applications (FPL'13)*, 2013, pp. 1–8.

[18] E. Lbbers and M. Platzner, "ReconOS: Multithreaded programming for reconfigurable computers," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 9, pp. 8:1–8:33, October 2009.

[19] T. Huffmire, S. Prasad, T. Sherwood, and R. Kastner, "Policy-driven memory protection for reconfigurable hardware," in *European Symposium on Research in Computer Security*, vol. LNCS 4189. Springer, September 2006, pp. 461–478.