Proof-carrying Hardware: Towards Runtime Verification of Reconfigurable Modules

Stephanie Drzevitzky, Uwe Kastens, Marco Platzner
University of Paderborn, Germany
stephanie.drzevitzky@upb.de

Abstract—**Dynamically reconfigurable hardware combines hardware performance with software-like flexibility and finds increasing use in networked systems. The capability to load hardware modules at runtime provides these systems with an unparalleled degree of adaptivity, but at the same time poses new challenges for security and safety.**

In this paper, we present proof-carrying hardware (PCH) as a novel approach to reconfigurable system security. PCH takes a key concept from software security, known as proof-carrying code, into the reconfigurable hardware domain. We outline the PCH concept and discuss runtime combinational equivalence checking as a first verification problem applying the concept. We present a tool flow and experimental results demonstrating the feasibility and potential of the PCH approach.

I. INTRODUCTION

Dynamically reconfigurable hardware combines hardware performance with software-like flexibility and finds increasing use in networked systems. The dynamic reconfiguration capability provides networked systems with the flexibility to download new hardware functionality or hardware updates as needed. Since system downtimes are often unacceptable, newly received hardware modules would have to be installed and run without previous extensive system testing and verification. However, due to its wide applicability reconfigurable hardware is often used in security and safety critical systems where an unintended system behavior could have severe consequences, such as heavy financial damage, loss of human life, threat to national security, etc. Assuring the absence of unwanted and malicious behavior caused by outside attacks as well as internal construction flaws becomes essential in such scenarios.

Security aspects for dynamically reconfigurable hardware have gained interest only recently. Kastner and Huffmire [1] present an overview of security risks present in every step of the life cycle of reconfigurable hardware. Drimer [2] also provides an exhaustive survey of possible attacks, involved parties, stages of life cycles, and defenses.

A first step towards security of reconfigurable hardware is to establish trust in the bitstream transmission. Typically, FPGA bitstreams are minimally secured by checksums and some FPGA vendors even offer built-in hardware support for bitstream decryption and embedded keys. Chaves et al. [3] propose a more flexible approach based on hashing to secure correct bitstream delivery. The same authors also address another aspect of reconfigurable hardware security: they interpret the incoming bitstream to check whether the physical

regions on the FPGA that are to be reconfigured match the intended reconfiguration area. Drimer and Kuhn [4] distinguish between authentication and confidentiality and discuss a security protocol that combines both aspects to prevent system downgrades. None of these techniques, however, looks at functional properties of reconfigurable modules.

Huffmire et al. [5] focus on multi-core reconfigurable systems where several cores access the same memory. The authors define a set of high-level policies covering security scenarios such as Chinese Wall and Secure Hand-Off. The designer uses a formal regular language to describe valid memory accesses for a core and the interactions included in the policies. Starting from this formal specification, a synthesis tool generates a memory reference monitor. In [6], Huffmire et al. propose to secure IP cores on FPGAs with physical isolation primitives called moats and drawbridges. While moats prevent unwanted and unanticipated communication between cores, drawbridges allow for controlled and secure communication channels. Drawbridges are especially useful when also a reference monitor is invoked which enforces a memory policy specified for the intended access scenario. The combination of physical isolation primitives and reference monitors, and the threats addressed by these techniques are also discussed in [7], [8].

Many methods known from the domains of software and (static) hardware verification rely on model checking (see, e.g., [9], for a survey). Since model checking techniques are rather runtime and resource consuming their applicability to runtime verification of reconfigurable hardware modules needs to be investigated yet. In this context, Singh and Lillieroth [10] propose the Core Verification Flow which conducts a decomposition of the core and extracts a reference design for each entity to be verified from the core's behavioral specification. Eventually, they receive a logical formula against which the implementation is compared with NP-Tools or Prover Software. In contrast to our approach, the Core Verification Flow runs the complete analysis and security verification on the system that also executes the core.

The novel contribution of this paper is the presentation of *proof-carrying hardware (PCH)* as an approach to reconfigurable system security. PCH takes a key concept from software security, known as proof-carrying code [11], into the reconfigurable hardware domain. With PCH, we shift the burden to create a proof for a desired security property from the consumer of a reconfigurable module to its producer. The producer has to invest the substantial computational resources

required to create a proof. The proof is then combined with the reconfigurable module into a binary and delivered as *proof-carrying bitstream*. The consumer, i.e., the target system, can quickly verify the security property and, if successful, instantiate and run the hardware module. In essence, the consumer is enabled to only run verified hardware modules without having to trust the producer or rely on a secured transmission process. The PCH concept is rather general and can be applied to many types of security properties and corresponding proof systems and proofs. As a first application of PCH, we focus on runtime combinational equivalence checking [12] in this paper.

The paper is organized as follows. In Section II, we discuss the novel concept of PCH. Section III elaborates runtime combinational equivalence checking as a first PCH application and shows the required functionalities for producers and consumers of reconfigurable modules. The implementation of a complete tool flow and experimental results are depicted in Section IV. We conclude the paper and point to future work in Section V.

II. From Proof-carrying Code to Proof-carrying Hardware

In 1996, Necula et al. proposed Proof-Carrying Code [11] as a means to validate code from an untrusted source before execution. The scenario includes a code consumer and a code producer. Both have to agree on a safety policy that includes formalisms to capture characteristics of the consumer's execution platform (e.g., machine model, memory accesses, type definitions) and to describe the desired safe code behavior. Depending on the actually chosen formalisms, producer and consumer must also decide on a corresponding proof system.

The producer then creates the program for the required functionality and, depending on the safety policy, may annotate the code with preconditions, invariants, and postconditions. This step is crucial, since the annotated code together with the safety policy is transformed into the so-called safety predicate. The safety predicate is then proven to hold true in all states of the program. The proof is combined with the code into the proof-carrying code delivered to the consumer. The consumer verifies the received proof and, thereby, checks the code's compliance with the safety policy.

The essence of proof-carrying code is to shift the burden of formal verification from the consumer to the producer, leaving the consumer with simply checking the delivered proof, a task of insignificant size compared to the actual computation of the proof. Thus, proof-carrying code techniques are of special interest for target systems (consumers) with limited computational resources or systems that need to quickly extend their functionality by downloading mobile code.

Proof-carrying code does not rely on secure transmission. Any damage to the proof section of the delivered code will result in a failed proof check at the consumer side. If tampering with the proof changes its content but does not render it incorrect, the proof will not match the code anymore which will also prevent its execution. Similarly, the code cannot be altered without failing to comply to the proof. In the unlikely case of matching changes to both the proof and the code,

there is no damage done since the proof still verifies the code and thus, guarantees its compliance with the safety policy. To cover the case that the safety policy has been altered without the consumer's knowledge before the proof is generated, the consumer can check whether the code annotations properly reflect his original safety policy. The only remaining requirement is a trustworthy procedure for checking the proof.

The great universality of the proof-carrying code approach has been demonstrated in several case studies. For example, in [11] a subroutine scans incoming network packets and determines whether they should be accepted as being valid or not. The safety policy has to secure memory safety and guarantee the termination of the subroutine. A further case study computing the checksum of an IP packet extends the concept to loops dealing with memory arrays of varying sizes. Another application of proof-carrying code is demonstrated in [13], where the safety policy not only captures memory access but also handles resource usage, e.g. execution times of agents.

The fundamental principle behind proof-carrying code states that for many computational problems validating a given solution is less complex and thus less costly in terms of computations than creating the solution in the first place. From this perspective, the proof-carrying code concept can be extended beyond its original, narrow definition and be applied to properties of interest other than safety features. Furthemore, a solution does not necessarily have to carry a proof in the literal meaning. For example, Klohs and Kastens [14] have applied the proof-carrying code concept to dataflow analysis. Later on, Klohs [15] extended the approach to intra- and inter-procedural program analysis.

We propose proof-carrying hardware as the equivalent of proof-carrying code for dynamically reconfigurable hardware systems. Reconfigurable hardware systems often have limited computational resources, rely on a fast instantiation of newly downloaded hardware modules, and are increasingly security and safety critical. All this characteristics match exactly the characteristics of target systems for which proof-carrying code has been established. Similar to the software scenario, consumer and producer of reconfigurable hardware modules have to agree on a (more or less) abstract machine model including a formalism to express safety features, and on a corresponding proof system. The burden of creating the reconfigurable module and the proof is with the producer, while the consumer should be able to quickly verify the safety features for the received module.

There are, however, notable differences between software and hardware. Arguably the most important one is the complexity of the machine model. Software executes on instruction-set processors which allows us to abstract code as a sequence of instructions accessing a rather small set of microarchitectural components. In contrast, reconfigurable hardware modules utilize large numbers of spatially arranged (placed and routed) components. Furthermore, while downloaded code plugs into rather matured software ecosystems, there are no standardized or even commonly used execution environments for reconfigurable hardware modules. Lastly,
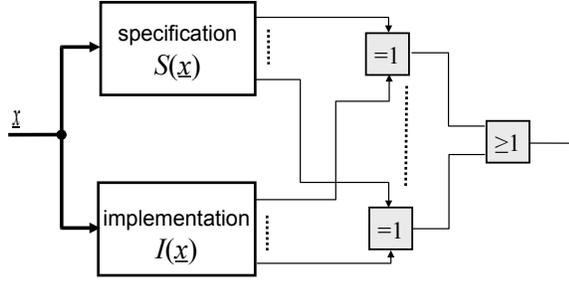
Fig. 1. Construction of the miter $M(S(\underline{x}), I(\underline{x}))$

while security and safety concepts for processor-based systems have been studied for quite some time, the field of reconfigurable hardware security is just emerging.

As its software counterpart, proof-carrying hardware (PCH) is a rather general and thus powerful concept. In the following section we will demonstrate runtime combinational equivalence checking as a first application example.

III. RUNTIME COMBINATIONAL EQUIVALENCE CHECKING

Combinational equivalence checking (CEC) is the most fundamental verification problem for hardware. The typical use of CEC is to verify whether a specification of a combinational function, $S(\underline{x})$, is equivalent to an implementation in a specific technology. To that end, the implemented circuit is analyzed and modeled with a logic function, $I(\underline{x})$. Apparently the number of inputs and outputs, respectively, of the specification and the implementation must match. Using $S(\underline{x})$ and $I(\underline{x})$, the miter is formed. The miter is a single-output function that provides both specification and implementation with the same inputs, and compares their outputs pairwise with XOR gates. All XOR outputs are then OR-ed together to form the miter, $M(S(\underline{x}), I(\underline{x}))$. Figure 1 shows the construction of the miter graphically.

If under any input $\underline{x}$ the specification and the implementation generate different outputs, the miter will evaluate to 1. Consequently, demonstrating equivalence means to prove the unsatisfiability of the miter. Modern CEC tools represent the miter in conjunctive normal form (cnf) and rely on Boolean satisfiability (SAT) solvers to prove unsatisfiability.

Interestingly, during the last years SAT solvers have progressed into tools that can generate resolution proofs for unsatisfiability. The motivation for this development roots in the desire to build up trust in the results generated by a SAT solver. The direct way would be to prove the correctness of a SAT solver itself, which unfortunately seems to be out of reach given the complexity of modern SAT solvers. The next best approach is to verify the unsatisfiability result for each single cnf, a step which requires access to a resolution proof. A resolution proof is a sequence of resolutions on the original cnf and intermediate clauses that eventually leads to an empty clause which models a contradiction. As the size of the generated proof has been a concern, proof traces have been proposed as a compact representation of a proof.

As an example, the following table gives a possible proof trace for the cnf $(x_1 + \overline{x}_2 + \overline{x}_3) \cdot (x_1 + x_3) \cdot (\overline{x}_1) \cdot (x_2 + \overline{x}_3)$:

(1)	$x_1 + \overline{x}_2 + \overline{x}_3$	
(2)	$x_1 + x_3$	
(3)	$\overline{x}_1$	
(4)	$x_2 + \overline{x}_3$	
(5)	x_3	using (2), (3)
(6)	x_2	using (4), (5)
(7)	$\emptyset$	using (5), (1), (6), (3)

The first four lines list the clauses of the cnf. Lines five to seven present the resolution steps and refer to the clauses used to resolve the new terms.

We can make use of resolution proof traces to set up a proof-carrying hardware scenario for runtime (online) CEC. Figure 2 shows the scenario and details the steps producer and consumer perform for runtime CEC. During runtime, the consumer decides to load a new reconfigurable hardware module with the combinational function $S(\underline{x})$ and sends a corresponding request to a producer. Classically, the producer will run the specification through logic synthesis tools which include FPGA technology mapping, and FPGA backend synthesis tools which include place & route and bitstream generation.

Additionally, the producer forms a miter from the specification $S(\underline{x})$ and the implementation $I(\underline{x})$ (synthesized netlist). A CEC tool proves the equivalence and generates the resolution proof trace $P(M(S(\underline{x}), I(\underline{x})))$. Finally, the producer combines the bitstream and the proof trace into the proof-carrying bitstream $B(\underline{x})$ and sends it to the consumer.

The consumer takes the received bitstream $B(\underline{x})$ and extracts the implemented logic function $I(\underline{x})$. After forming the miter with this implementation and the original specification, the consumer checks the proof using the proof trace. Only in case the proof holds, the bistream $B(\underline{x})$ is loaded into a reconfigurable area of the target device.

Any tampering with the bitstream or the proof sections of the proof-carrying bitstream will result in a failed proof check at the consumer side. If both are modified compatibly, the proof check will still fail since $I(\underline{x})$ does not correspond to the original specification $S(\underline{x})$ anymore.

In abstract terms, the safety policy includes the agreement on a specific bitstream format, on cnf to represent combinational functions, and on the use of propositional calculus with its resolution rules to derive and verify the proof. Furthermore, since the resolution proof indexes the clauses of the miter, consumer and producer must use the same way of constructing the miter.

IV. PROTOTPYE IMPLEMENTATION AND RESULTS

To demonstrate the feasibility of runtime CEC (as an application of PCH) we set up the prototype tool flow shown in Figure 3. The proof-of-concept tool flow bases on a number of freely available, non-commercial CAD tools which gives
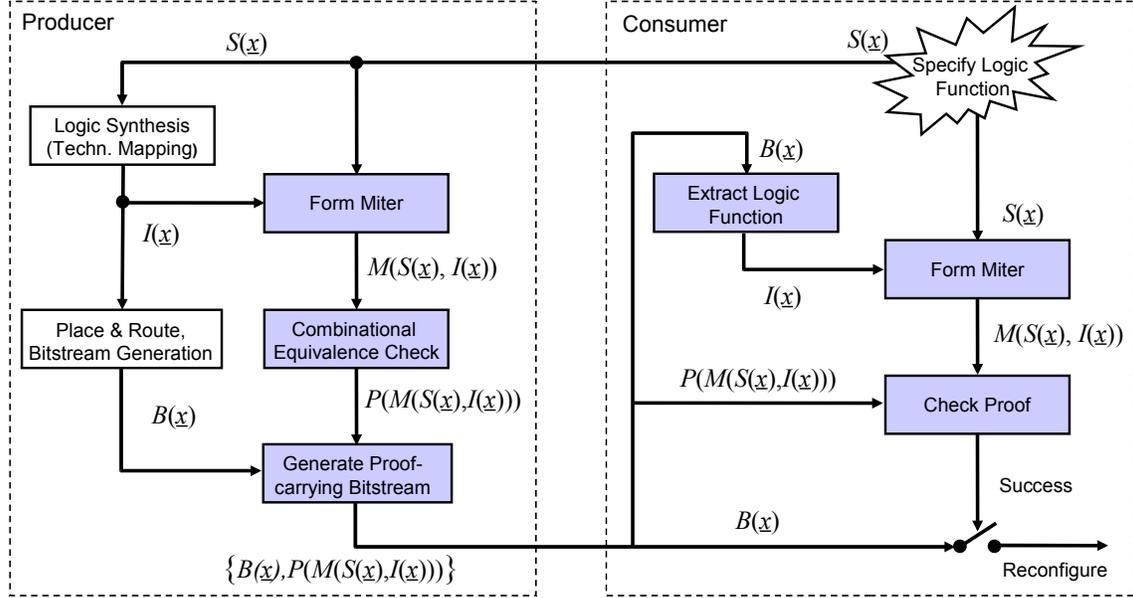
Fig. 2.   PCH scenario for runtime combinational equivalence checking

us full control over the different parameters involved and file formats used. The consumer describes a combinational test function in Verilog and sends it to the producer. Then, the following tools are invoked:

- Odin [16], [17] performs front-end synthesis and translates the incoming circuit specification in Verilog into a logic description in blif (Berkeley Logic Interchange Format).
- ABC [18] performs technology mapping to 4-LUTs, generating again a blif file.
- T-VPack [19], [20] packs the LUTs into clustered logic blocks, generating a circuit description in form of a netlist (.net) of such logic blocks. The experiments described in this paper use non-clustered logic blocks, but run T-VPack to convert the LUT netlist into the format required by VPR.
- VPR [19], [20] places and routes the packed logic block netlist and produces placement (.p) and routing (.r) information for inclusion in the final bitstream.
- To form the miter cnf, we use again ABC.
- The SAT solver PicoSAT [21] proves the unsatisfiability of the miter and generates a resolution proof trace.
- TraceCheck [22] at the consumer side validates the correctness of the proof trace and thereby gives the final approval to load and execute the received bitstream.

Our prototype setup uses Verilog to describe the combinational function at the consumer just for simplicity. In principle, any other and perhaps simpler specification formalism for capturing combinational functions can be used. In case the set of functions that might be requested as reconfigurable modules is known beforehand, the consumer will store the corresponding specifications rather than generating them at runtime.

We test the prototype tool flow on an Intel Core 2 Duo 2GHZ CPU with 4GB RAM running Linux 2.6.27.25-0.1 and report on results using the following test functions: a 128-bit parity function, n-bit combined adder/subtractors with $n = 8$ and $n = 128$, respectively, and an EBCDIC to ASCII converter for the letter subset of the EBCDIC code. The resulting FPGA bitstreams implement all test functions with LUTs; special circuitry such as carry chains or heterogeneous blocks are not yet included in our FPGA model.

Using the prototype tool flow and the test functions we experimentally investigate the following questions:

1) Does the runtime CEC tool flow work correctly?
2) What are the runtimes for producer and consumer? The difference between the time required for proving the miter unsatisfiable and generating the proof trace versus the time required to check the proof is of special interest. This time difference constitutes the main savings for the consumer, in comparison to an approach where the consumer runs the complete verification tool chain.
3) How large is the overhead for the proof-carrying bitstream? The proof trace increases the size of the bitsream and leads to higher transmission costs.

As a first result, we are able to demonstrate the correct functionality of the runtime CEC prototype. The producer generates FPGA implementations and equivalence proofs for all test functions, which the consumer successfully verifies. Tampering with the files at different stages of the producer tool flow results in the expected effects: Modifying the technology-mapped netlist of the miter cnf or the miter cnf itself results in a satisfiable miter, stating that the implemented circuit differs from the specification. Changing the proof trace at the
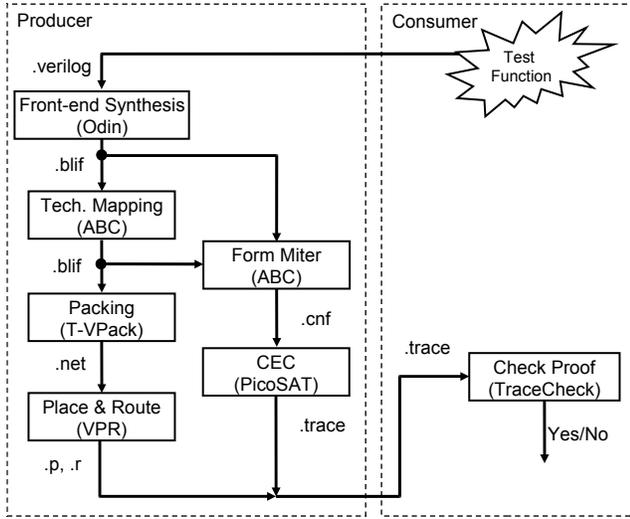
Fig. 3. Runtime CEC prototype tool flow

Test function	Size of cnf [Vars×Clauses]	Producer		Consumer
		Overall [s]	PicoSAT [s]	TraceCheck [s]
8-bit add/sub	104×494	1.176	0.012	0.004
128-bit add/sub	1625×8271	32.134	0.360	0.076
128-bit parity	130×2	2.216	0.008	0.004
converter	137×562	3.880	0.008	0.004

TABLE I
RUNTIMES MEASUREMENTS FOR PRODUCER AND CONSUMER IN THE
ONLINE CEC SCENARIO.

consumer or during transmission leads to a failed proof check at the consumer.

The prototype tool flow shown in Figure 3 covers the main components and functionalities of the runtime CEC scenario. We want to re-emphasize that the proof-carrying hardware concept does not require us to assess or verify the correctness of the producer's tools. However, in our current prototype the consumer does not yet form the miter from the received bitstream and the original specification. As a result, the runtime CEC cannot detect an incorrect operation of the front-end synthesis tool (Odin) or simultaneous and matching modifications of the bitstream and the corresponding proof. The missing components, which are shown in Figure 2, are the creation of the proof-carrying bitstream at the producer and the extraction of the implemented logic function from the received bitstream and the forming of the miter at the consumer. The implementation of these components is future work. The bitstream creation and extraction of the logic function are straight-forward tasks. The formation of the miter from the extracted logic function and the specification will be done by ABC in the same way as at the producer's side, and allow the consumer to increase the trust level.

An important metric for runtime CEC is the required computation time, especially for the consumer. We measure the runtimes of all tools in our prototype. On the producer side, this includes Odin and ABC for creating the blif file as well as the miter cnf, T-VPack and VPR for packing, place and route, and PicoSAT for equivalence checking and proof trace generation. On the consumer side, we measure the runtime of TraceCheck. Table I presents the results for our test functions. Column two of Table I gives the complexity of the resulting miter cnf in number of variables and clauses. Columns three and four report the overall producer runtime and the time required for PicoSAT, and column five reports the consumer runtime. All times are CPU times, adding up user and system

process times.

As Table I shows, the consumer runtime is three orders of magnitude lower than the producer runtime. Interestingly, for our test functions the producer spends only a marginal fraction of its overall runtime in the SAT solver. Future work needs to experimentally investigate the runtime behavior for larger circuits and circuits known to be difficult to verify. Of special interest is the point from which on the SAT solver begins to dominate the producer's overall runtime.

The key point of proof-carrying hardware, however, is the low computational effort for the consumer. To further underline this point, Table II compares the runtimes of the SAT solver (PicoSAT) and the proof checker (TraceCheck) for cnf problems from the 2008 SAT-Race_TS_1 benchmark. The cnfs differ greatly in the number of variables and clauses, but include instances with up to six orders of magnitude more variables and clauses (`narai_vpn-10s`) than the miters for our test functions. The last column of Table II shows that the effort for checking a proof trace is between one and three (five in an extreme case) orders of magnitude lower than for proving equivalence and generating the proof trace.

cnf instance	Size of cnf [Vars×Clauses]	SAT [s]	Check [s]	Factor
een-tip-sr06-par1	163647×484827	6.412	0.024	267
een-tipb-sr06-tc6b	40196×115775	3.028	0.012	252
godlb-heqc-desmul	28902×179895	75.697	1.512	50
goldb-heqc-rotmul	5980×35229	31.458	2.272	13
hooons-vbmc-s04-05	8503×25097	11.065	1.536	7
hooons-vbmc-s04-07	25900×77627	158.294	8.557	18
manol-pipe-c10b	43517×129265	87.937	1.075	81
manol-pipe-c10ni_s	204664×609478	255.584	0.004	63896
manol-pipe-c6id	82022×242044	5.460	0.052	105
manol-pipe-c6n	37147×110077	49.547	0.820	60
manol-pipe-c6nid_s	148051×438562	5.528	0.020	276
manol-pipe-c7_i	13023×38509	17.685	0.292	60
manol-pipe-c7idw	112620×333058	131.444	1.412	93
manol-pipe-c8_i	14052×41596	74.433	2.028	36
manol-pipe-c8b_i	32057×95005	13.485	0.256	52
manol-pipe-c8n	53697×159595	111.351	1.988	56
manol-pipe-f6b	37002×109570	6.672	0.328	20
manol-pipe-f6n	37452×110920	7.500	0.236	31
manol-pipe-g10idw	174122×516784	141.241	1.964	71
manol-pipe-g6bid	40371×118192	5.272	0.112	47
manol-pipe-g7n	23936×70492	5.784	0.248	23
narai_vpn-10s	2270930×8901946	306.335	0.368	832
schup-l2s-s04-abp4	14809×48429	67.528	7.896	8
velev-npe-1.0-02	3295×35407	23.577	3.748	6
velev-sss-1.0	1453×12526	20.741	3.744	5

TABLE II
RUNTIME COMPARISON BETWEEN PICOSAT (SAT) AND TRACECHECK
(CHECK) FOR BENCHMARKS OF THE 2008 SAT-RACE_TS_1.

Test function	Raw bitstream size [KByte]	Proof trace size [KByte]	Overhead [%]
8-bit add/sub	24.4	28.1	115.2
128-bit add/sub	458.7	835.5	182.2
128-bit parity	87.7	31.0	35.4
converter	131.0	20.1	15.3

TABLE III
SIZE MEASUREMENTS FOR THE RAW BITSTREAM AND PROOF TRACE.

We further provide an estimate on the overhead incurred by adding a proof trace to the delivered bitstream. The accuracy of the estimate is limited due to two facts. First, to measure the overhead in dependence of the circuit's size we operate VPR in the area minimization mode. For each circuit, VPR chooses a logic block array just large enough to accommodate the circuit and a channel width just sufficient to route the circuit. We denote the size of the resulting netlist, including placement and routing information, as an estimate for the size of the raw bitstream to which we add the proof trace. Second, our tool flow does not yet specify a binary bitstream format. Hence, we measure the sizes of the corresponding text files. Table III lists the size of the raw bitstream, the proof trace, and the corresponding overhead for our test functions. The results show a wide variation in overheads from the rather low 15% for the EBCDIC-ASCII converter to some 180% for the larger arithmetic circuit. Although these numbers are overly pessimistic and coarse estimates, they point to the need of investigating a binary format for the proof-carrying bitstream using compression techniques.

V. CONCLUSION AND FUTURE WORK

We present proof-carrying hardware (PCH) as a novel approach to reconfigurable system security and demonstrate its potential by applying it to runtime combinational equivalence checking (CEC). We elaborate on a prototype tool flow for runtime CEC and discuss experimental results. Using proof traces generated by a modern SAT solver, the prototype clearly indicates a significant reduction in computational cost for a reconfigurable target system.

The prototype tool flow presented in this paper is a first step in exploring the potential of PCH. As a next step we will complete the tool flow shown in Figure 2. Future work also includes extending VPR to also enforce physical (structural) isolation primitives such as moats and drawbridges [6].

Due to its generality and flexibility, the PCH concept can have a great impact on reconfigurable hardware security. The ultimate goal is to perform runtime verification of entire bitstreams, i.e., proving a reconfigurable module structurally and functionally correct with only minimal assumptions about the correctness of tools, and without having to reason about trust levels for code producers and third parties.

ACKNOWLEDGMENT

REFERENCES

[1] R. Kastner and T. Huffmire, "Threats and Challenges in Reconfigurable Hardware Security," in *International Conference on Engineering of Reconfigurable Systems & Algorithms (ERSA)*. CSREA Press, July 2008.

[2] S. Drimer, "Volatile FPGA design security–a survey," Computer Laboratory, University of Cambridge, 2008.

[3] R. Chaves, G. Kuzmanov, and L. Sousa, "On-the-fly Attestation of Reconfigurable Hardware," in *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, September 2008, pp. 71–76.

[4] S. Drimer and M. Kuhn, "A Protocol for Secure Remote Updates of FPGA Configurations," in *Reconfigurable Computing: Architectures, Tools and Applications (ARC)*, vol. 5453. Springer, 2009, pp. 50–61.

[5] 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.

[6] T. Huffmire, B. Brotherton, G. Wang, T. Sherwood, R. Kastner, T. Levin, T. Nguyen, and C. Irvine, "Moats and Drawbridges: An Isolation Primitive for Reconfigurable Hardware Based Systems," in *Symposium on Security and Privacy*. IEEE, 2007, pp. 281–295.

[7] T. Huffmire, B. Brotherton, N. Callegari, J. Valamehr, J. White, R. Kastner, and T. Sherwood, "Designing Secure Systems on Reconfigurable Hardware," *ACM Transactions on Design Automation of Electronic Systems*, vol. 13, no. 3, pp. 1–24, July 2008.

[8] T. Huffmire, B. Brotherton, T. Sherwood, R. Kastner, T. Levin, T. Nguyen, and C. Irvine, "Managing Security in FPGA-Based Embedded Systems," *IEEE Design & Test of Computers*, vol. 25, pp. 590–598, November/December 2008.

[9] V. D'Silva, D. Kroening, and G. Weissenbacher, "A Survey of Automated Techniques for Formal Software Verification," in *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7. IEEE, July 2008, pp. 1165–1178.

[10] S. Singh and C. Lillieroth, "Formal Verification of Reconfigurable Cores," in *Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 1999, pp. 25–32.

[11] 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.

[12] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-Aware AIG Rewriting: A Fresh Look at Combinational Logic Synthesis," in *Design Automation Conference (DAC)*. ACM, July 2006, pp. 532–535.

[13] G. Necula and P. Lee, "Safe, Untrusted Agents Using Proof-Carrying Code," in *Mobile Agents and Security*, vol. LNCS 1419. Springer, 1998, pp. 61–91.

[14] K. Klohs and U. Kastens, "Memory Requirements of Java Bytecode Verification on Limited Devices," in *Proceedings of the 3rd International Workshop on Compiler Optimization Meets Compiler Verification (COCV)*, 2004.

[15] K. Klohs, "A Summary Function Model for the Validation of Interprocedural Analysis Results," in *Proceedings of the 7th International Workshop on Compiler Optimization meets Compiler Verification (COCV)*, 2008.

[16] "Odin A Verilog RTL Synthesis Tool for Heterogeneous FPGAs." [Online]. Available: http://www.eecg.toronto.edu/~jayar/software/odin/index.html

[17] P. Jamieson and J. Rose, "A Verilog RTL synthesis tool for heterogeneous FPGAs," in *International Conference in Field Programmable Logic and Applications (FPL)*, August 2005, pp. 305– 310.

[18] *Going Places with ABC*, Berkeley Logic Synthesis and Verification Group. [Online]. Available: http://www.eecs.berkeley.edu/~alanmi/abc/abc_tutorial.ppt

[19] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *International Conference on Field Programmable Logic and Applications (FPL)*, vol. 1304. Springer, 1997, pp. 213–222.

[20] V. Betz, "VPR and T-VPack User's Manual (Version 5.0)," 2008.

[21] A. Biere, "PicoSAT Essentials," in *Journal on Satisfiability , Boolean Modeling and Computation*, vol. 4, 2008, pp. 75–97.

[22] C. Sinz and A. Biere, "Extended Resolution Proofs for Conjoining BDDs," in *1st Intl. Computer Science Symp. in Russia*, vol. 3967. Springer, 2006.