Proof-Carrying Hardware: A Novel Approach to Reconfigurable Hardware Security

Der Fakultät für Elektrotechnik, Informatik und Mathematik der Universität Paderborn

zur Erlangung des akademischen Grades Doktor der Naturwissenschaften (Dr. rer. nat.)

vorgelegte Dissertation von

Stephanie Drzevitzky
geboren am 30.12.1981 in Köln

Zusammenfassung

FPGAs, System on Chips und eingebettete Systeme sind heutzutage kaum mehr wegzudenken. Sie kombinieren die Rechenleistung von spezialisierter Hardware mit einer Softwareähnlichen Flexibilität. Zur Laufzeit können sie ihre Funktionalität anpassen, indem sie online neue Hardware Module beziehen und deren Funktionalität integrieren. Mit der Leistung wachsen auch die Anforderungen an rekonfigurierbare Hardware. Ihr Einsatz in immer sicherheitskritischeren Szenarien erfordert neue Wege um Sicherheit zu gewährleisten, da ein Versagen der Sicherheit gravierende Folgen mit sich bringt. Neben finanziellen Verlusten sind auch der Verlust von Menschenleben oder Einbußen in der nationalen Sicherheit denkbar.

In dieser Arbeit stelle ich das neue und wegweisende Konzept der beweistragenden Hardware vor. Es ist eine Methode zur Verifizierung von Eigenschaften von Hardware Modulen um die Sicherheit der Zielplatformen zur Laufzeit zu garantieren. Der Produzent eines Hardware Moduls liefert, basierend auf den Sicherheitsbestimmungen des Konsumenten, einen Beweis der Sicherheit mit dem Rekonfigurierungsbitstrom. Die aufwendige Berechnung des Beweises steht im Kontrast zu der vergleichsweise unaufwendigen Überprüfung durch den Konsumenten. Ich präsentiere einen Prototypen basierend auf Open Source Werkzeugen und einer eigenen abstrakten FPGA Architektur samt Bitstromformat. Den Nachweis über die Nutzbarkeit von beweistragender Hardware erbringt die Evaluierung des Prototypen zur beispielhaften Anwendung der Sicherung von kombinatorischer und begrenzt sequenzieller Äquivalenz von Referenzmonitor-Modulen zur Speichersicherheit.

Abstract

FPGAs, systems on chip and embedded systems are nowadays irreplaceable. They combine the computational power of application specific hardware with software-like flexibility. At runtime, they can adjust their functionality by downloading new hardware modules and integrating their functionality. Due to their growing capabilities, the demands made to reconfigurable hardware grow. Their deployment in increasingly security critical scenarios requires new ways of enforcing security since a failure in security has severe consequences. Aside from financial losses, a loss of human life and risks to national security are possible.

With this work I present the novel and groundbreaking concept of proof-carrying hardware. It is a method for the verification of properties of hardware modules to guarantee security for a target platform at runtime. The producer of a hardware module delivers based on the consumer's safety policy a safety proof in combination with the reconfiguration bitstream. The extensive computation of a proof is a contrast to the comparatively undemanding checking of the proof. I present a prototype based on open-source tools and an abstract FPGA architecture and bitstream format. The proof of the usability of proof-carrying hardware provides the evaluation of the prototype with the exemplary application of securing combinational and bounded sequential equivalence of reference monitor modules for memory safety.

Contents

List of Tables

Listings

List of Figures

CHAPTER 1

Introduction

1.1 Thesis Context

Our everyday lives are dominated by desktop computers, smartphones, navigation devices for cars and other appliances. They adjust to our every need by downloading more information and more tools, i.e. apps and programs, from the internet and even update themselves, for instance with new maps. Gone are the times of updates delivered on CDs, updates now occur whenever needed. But not only private computer systems have undergone this change, as a matter of fact most computer systems are networked nowadays and benefit from the flexibility to update. Every alteration of a computer system's functionality is a potential security threat. Since the software-ecosystem that exists around the instruction set architecture is well-known and the domain of software security has been researched quite extensively, the risks of software updates can be contained.

Reconfigurable devices and in particular dynamically reconfigurable devices have in recent years gained importance. They offer the performance of dedicated hardware and combine it with a level of flexibility that is usually only offered by software. With growing capacities, reconfigurable devices such as Field-Programmable Gate Arrays (FPGAs), are increasingly deployed in varying scenarios. The variety of scenarios grows and becomes more challenging up to the point where embedded systems are deployed in unknown environments with unknown security risks. The failure to provide security and safety in those scenarios would have severe consequences; such as financial losses, a loss of human life, and threats to national security. Security for reconfigurable hardware and dynamically reconfigurable hardware is a novel research area that has only been emerging for the last few years but as reconfigurable devices such as FPGAs gain importance, security for reconfigurable devices gains importance as well. Especially in the very recent years, research for reconfigurable hardware security has taken a step forward. Hardware attacks, such as physical attacks but also side channel attacks and Trojan horses, countermeasures, and FPGA and embedded system security have been investigated, see [37, 8] but also a

taxonomy to classify the countless threats to reconfigurable hardware security, see [63].

One important aspect of security for reconfigurable devices is the reconfiguration process. Reconfigurable devices offer points of attack through the reconfiguration itself. Several approaches at security aim at securing the transmission of the reconfiguration bitstream, often with the aide of encryption, as will be elaborated in Chapter 2, to deliver a confidential and authentic bitstream. A secure transmission can only be a first step to secure reconfiguration as the new component's impact on the system's integrity is of major importance. Other approaches give assurances of single aspects of system security after reconfiguration but still do not concern themselves with the actual functionality of the newly installed hardware module.

Another important vulnerability of the reconfiguration process is the source of the reconfiguration bitstream, i.e. the producer. The production of a reconfiguration bitstream (as well as a reconfigurable device) typically involves multiple parties and stages. Usually, a consumer established trust in the final supplier who then trusts his suppliers. Even if trust can be established in all parties involved, the final reconfigurable device in combination with the new bitstream is more than the sum of its parts. Trust in the final product poses a new security challenge. Despite his best intentions, a trusted supplier may deliver a bitstream that corrupts its host platform with unintended side-effects.

Considering that security critical scenarios can demand an absolute guarantee of certain properties of the delivered hardware module, it is not enough to establish trust in a bitstream producer. Security for reconfigurable devices means a secure reconfiguration bitstream whose specific properties and features are known before execution. Considering that producers deliver hardware modules in form of reconfiguration bitstreams on demand, it seems only logical that they should deliver the required security assurances as well. A new approach to bring runtime verification of bitstreams from untrusted sources to dynamically reconfigurable devices will be introduced in this work.

1.2 Thesis Contribution

In this thesis, I introduce the novel concept of proof-carrying hardware. It is an approach to runtime verification of hardware modules from untrusted sources based on the security guarantees regarding the new module's functional properties to establish on-the-fly trust and on-the-fly reconfiguration of reconfigurable devices.

Three key aspects distinguish this approach from other approaches in the broader field of hardware security:

- Firstly, it enables reconfigurable devices such as FPGAs to benefit from formal verification without high computational effort. The producer of the hardware module includes a safety proof in the bitstream which the consumer then only has to check.
- Secondly, this approach is flexible. Proof-carrying hardware can adjust to the verification of more than one security property of a hardware module and can be employed for multiple use cases.

- Thirdly, proof-carrying hardware is robust as the safety proof is based on and later matched against the safety policy determined by the consumer of the bitstream. This guarantees that tampering with the bitstream by either the producer or a third party would result in a failure to conform to said safety policy.

In this thesis, I also present a prototype proof-carrying hardware tool flow. The tool flow demonstrates the application of proof-carrying hardware to the runtime verification of combinational equivalence and bounded sequential equivalence. These are used on reference monitor hardware modules, compiled from memory access polices to manage memory access by multiple other IP cores on the fabric. This groundwork is meant as a basis for later extension.

I also deliver a feasibility study based on the prototype tool flow. The study clearly demonstrates the successful shift of workload from the consumer to the producer, meaning that I successfully demonstrated the benefit of this novel technique for reconfigurable platforms.

1.3 Thesis Structure

This thesis is structured as follows:

Chapter 2 elaborates the background of this work and related work in the area of security for reconfigurable devices. An introduction to Field-Programmable Gate Arrays (FPGAs) includes an introduction to the physical aspects of the device as well as the reconfiguration process which equips the FPGA with a new functionality. To give more context in which to place this thesis, I give an overview of other approaches to remedy some of the risks that are inherent to dynamic reconfiguration of FPGAs. For a better understanding of proof-carrying hardware, I elaborate on proof-carrying code, a concept for guaranteeing security properties of software modules introduced in 1996 by Necula and Lee [57] which is crucial for the understanding of this work.

Chapter 3 compares software security with security for reconfigurable hardware and the difficulties inherent to any approach to secure reconfigurable hardware. The proof-carrying hardware approach is introduced and its key concepts are emphasized. I then compare my novel approach to already existing works in the field of reconfigurable hardware security. The chapter concludes with the thesis claim.

Chapter 4 details the proof-carrying hardware approach to runtime verification. It explains the different types of security challenges to which I later apply proof-carrying hardware: combinational equivalence checks of design and design specification, bounded sequential equivalence checks of design and design specification, and temporal isolation of memory ranges accessed by hardware modules on an FPGA chip. In this context, Chapter 4 discusses miter functions and resolution proofs used for the verification process as well as temporal isolation of memory achieved through reference monitor hardware modules based on dynamic and static types of memory access policies.

Chapter 5 details the evaluation methodology of this thesis which is a proof-carrying hardware open-source prototype tool flow. The tool flow is based on a novel abstract

FPGA architecture and an according bitstream format, both are elaborated. The open-source tools and formats are depicted. To implement the proof-carrying hardware principle of shifting the majority of the verification workload to the untrusted source producer of a hardware module, the tool flow realizes a split between producer and consumer. This separation and the resulting separation of tasks is demonstrated.

Chapter 6 gives an evaluation of the proof-carrying hardware approach based on the tool flow presented in the previous chapter. Firstly, the robustness and limitations of the implementation and proof-carrying hardware are discussed. To demonstrate the usability of the concept, measurements regarding runtime and memory usage document the desired shift of workload. Those results are discussed in detail.

Chapter 7 summarizes the contributions of this work as well as the results and conclusions drawn from them. This thesis finishes with an outlook on future work regarding proof-carrying hardware.

CHAPTER 2

Background and Related Work

This chapter gives an introduction into reconfigurable hardware and reviews existing work in the field of hardware security and also the concept of proof-carrying code from the software domain. This information serves a better understanding of this thesis and its contextual placement.

In Section 2.1, I give fundamental information on the make-up on reconfigurable devices and elaborate on the reconfiguration process. In Section 2.2, I discuss the matter of hardware security. The focus is on possible attacks on embedded systems such as FPGAs and their available and theoretical possible countermeasures. Section 2.3 gives an overview of Proof-Carrying Code, a security concept from the software domain which is fundamental for this thesis.

2.1 Reconfigurable Hardware

2.1.1 Field-Programmable Gate Arrays

Field-programmable gate arrays (FPGAs) are programmable logic devices, i.e. integrated circuits whose functionality is defined by the end user through the FPGA's programming. FPGAs consist of three basic components: I/O blocks for information flow between the FPGA and its environment, programmable logic blocks, and programmable interconnect. The logic blocks are arranged in arrays and programmed with the logic that makes up the FPGA's functionality. The interconnect uses routing resources of varying length to combine the logic blocks to greater functionality and connect them to I/O blocks, as shown in Figure 2.1.

Logic blocks are comprised of three key components: multiplexers, look-up tables (LUTs), and registers or flip-flops (FFs). The amount of the different components and their arrangement allows for the design of minimalistic models of FPGAs, see Figure 2.2, as well as complex commercial FPGAs, see Figure 2.3. Each logic block has input and

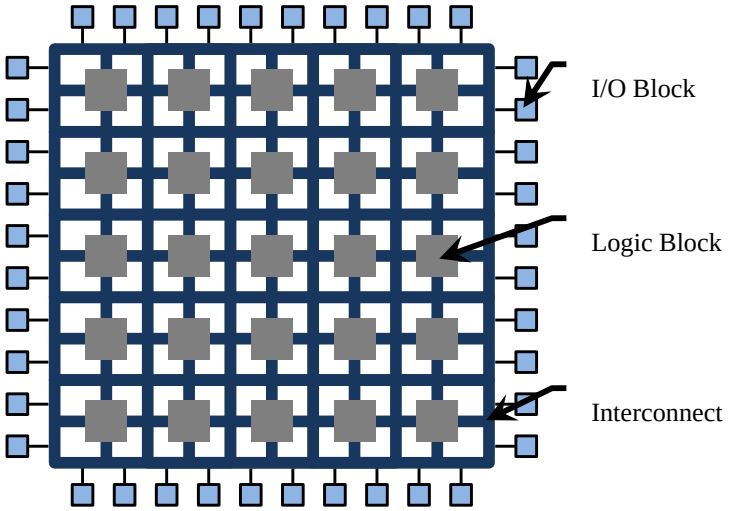


Figure 2.1: Generic FPGA.

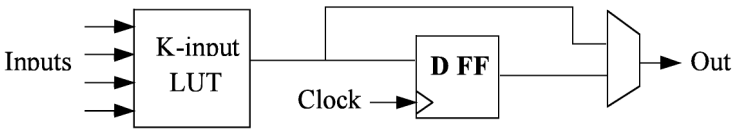


Figure 2.2: Logic block used in open-source FPGA Architecture by Betz et al., see [13]. The logic block contains each one FF, LUT, and MUX.

output capabilities that can connect to the routing resources surrounding the block via programmable connections. The routing itself consists of routing segments of varying length and switch boxes connecting the segments, see Figure 2.4. Shorter segments are intended for a direct connection between neighboring logic blocks while longer segments link components further apart on the device.

As stated above, the functionality of an FPGA is determined by its programming, i.e. configuration. A configuration bitstream contains the necessary values for the programmable parts that, after their respective programming, together add up to the desired functionality. The fabrication of any bitstream begins with the design of the hardware circuit in a hardware description language (HDL), e.g. Verilog and VHDL (Very High Speed Integrated Circuit Hardware Description Language). A work flow takes place between this description of the hardware functionality and the actual bitstream. Examples for such work flows or tool chains are the ones offered by Xilinx, Inc. and Altera Cooperation: the XST (Xilinx Synthesis Technology), see [42], and the Quartus II software, see [23], respectively. There are several main steps common to all work flows, as depicted in Figure 2.5:

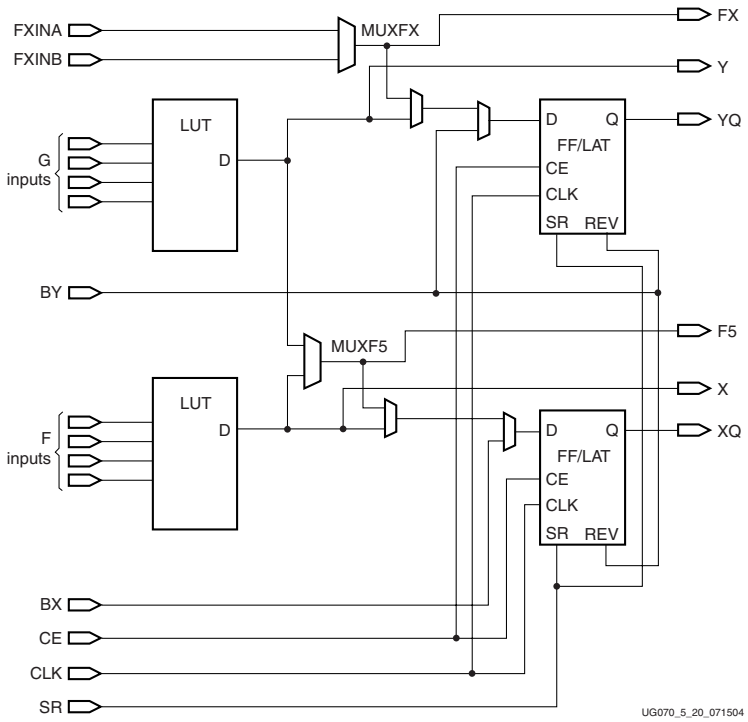


Figure 2.3: Simplified Virtex-4 FPGA logic block by Xilinx, Inc., see [41]. The logic block contains multiple FFs and LUTs connected by MUXs.

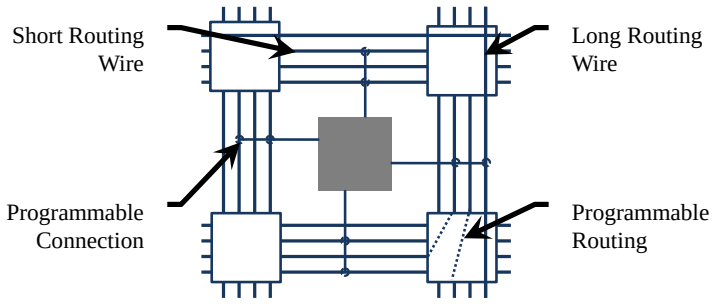


Figure 2.4: Example routing resources. Switch boxes connect incident wire segments.

- Synthesis and optimization of the design:
  The design is translated from HDL specification into a netlist format while optimizing the circuit under various aspects such as a minimum usage of logic blocks or reduction of redundant parts of the circuit.

- Technology Mapping:
  The netlist is mapped onto the actual hardware by representing the circuit with

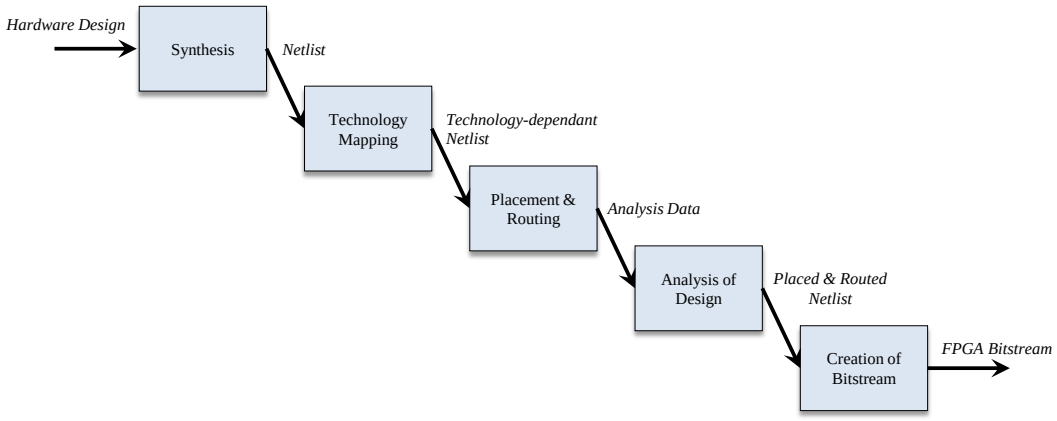


Figure 2.5: Steps of a generalized design tool flow.

the available gates and resources. This technology dependent netlist may be further optimized for delay and area minimization.

- Placement and routing of the design:
  The design undergoes a mapping to the physical device. The components of the hardware device, e.g. logic blocks, are allocated to the parts of the circuit and routing connections are defined.

- Analysis of placed and routed design:
  Various features of the implementation are analyzed according to demand. Common analysis foci are the number of routing resources of different length, the timing delay, the number of used logic blocks, etc.

- Creation of bitstream:
  The hardware functionality is translated into a format that the specific hardware device uses for reconfiguration. The bitstream also contains the information needed to map the new functionality to its physical components, e.g. information regarding placement and routing.

The first step of the above list is a translation or series of translations where the end result is a circuit in a netlist format. The circuit is now in a form that is compatible for placement and routing according to the physical layout of the reconfigurable device. The placement of the circuit assigns each logic block and possible other components of the device its functionality, the routing then programs the routing resources to connect all components on the FPGA. The placement and routing takes into account the physical properties of the reconfigurable fabric as well as user constraint, such as special timing requirements or area use.

Note that some work flows do not keep these steps completely separated and independent from another but have earlier steps in the work consider assignments of other tools

that are evoked at a later time in the work flow. That way, information which would otherwise be lost can be sensibly saved to benefit later tools or even enable operations that would not be possible or feasible without these information. One instance of this is the "Recording Synthesis History for Sequential Verification" introduced by Mishchenko and Brayton in [53] which promotes a synergy between synthesis and verification. The synthesis uses And-Invertor Graphs (AIG) which are circuits composed exclusively of two-input AND gates and invertor to negate signals. The synthesis process operates with two AIG managers where a Working AIG (WAIG) reflects the current state of the synthesis and a History AIG (HAIG) records every AIG nodes that were present in the circuit at any time. The result of this form of synthesis is a simpler and faster verification process.

The work flow then composes the bitstream from the data created so far and the information about the actual FPGA. The bitstream can be transferred to the FPGA over any interface the FPGA provides. Hence, remote updates via internet or network are as possible as direct uploads of the bitstream via cable interface. Once the reconfiguration cycle is completed, the FPGA functions as the designed hardware circuit. The basic reconfiguration of an FPGA assigns new content to all programmable parts of the entire device, such as logic blocks and interconnect. Some FPGAs offer a reconfiguration technique that allows to reconfigure only a selected section of the device. This partial reconfiguration is done at runtime as all unaffected sections keep operating while a separated part of the fabric is given a new functionality. The device is divided into static sections for this purpose, each of the sections can be individually configured. For further information on FPGA design flows see [15].

The reconfiguration of FPGAs is especially interesting when dealing with reconfigurable system on chips. A system on chip denotes an integrated circuit that combines many or all functions of an entire computer system on a single chip. In this context, FPGAs are considered reconfigurable systems on chip. Systems on chips have a special function, examples are the chips in smartphones, navigation devices for cars, and missile guidance systems. The reconfiguration of such an embedded system, a system on chip in combination with another technical device, is particularly appealing as it offers the chance to update devices after their deployment.

2.1.2 FPGA Architecture and Design Tool Research

FPGA architecture research is concerned with studying and determining the hardware components of FPGAs, i.e. the blocks for computation and routing, and their interconnects. Research on FPGA design tools involves the analysis and development of algorithmic methods for technology mapping, placement, routing and timing analysis. Architecture and design tools are interdependent: An architecture is needed as the basis for tool development, and design tools are needed to drive the development of architectures. For example, the evaluation of a specific architecture requires the placement and routing of benchmark circuits on that architecture. I shall elaborate the analysis of a placed and routed circuit with the example of the VPR (Versatile Place and Route) tool by Betz [13]: VPR is an open-source tool that takes a netlist and an FPGA architecture file as input

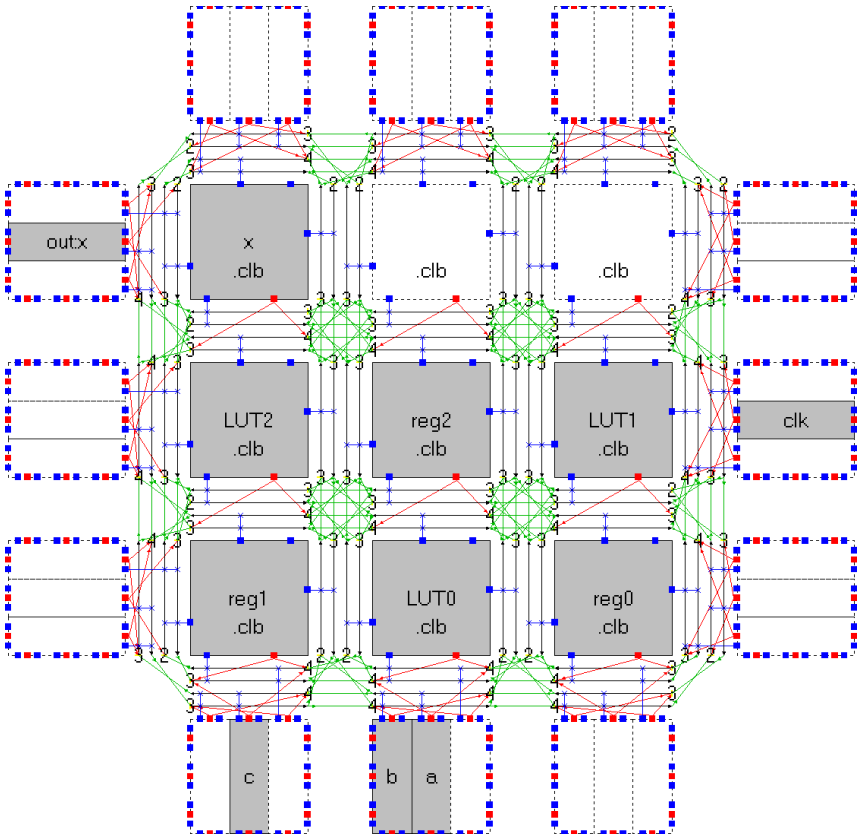


Figure 2.6: VPR generated circuit on FPGA chip with a 3×3 grid of configurable logic blocks and global I/O. The circuit uses the input variables a, b, c and a clock clk and returns the output variable x. It consists of three registers and four LUTs. Highlighted are all global I/O pads, the I/O pins of the clbs, and the routing resources such as switch boxes and connection boxes to connect all components.

and returns the placement and routing for the netlist according to the described FPGA architecture. VPR is an excellent open-source tool for FPGA architectural research as it offers an array of placement and routing options and returns analyses regarding the implementation.

The FPGA architecture file used for VPR lets the user specify many aspects of the physical layout of the FPGA fabric. The layout of the device and its general properties are designed by specifying, for example, how many rows and columns of logic blocks there should be, what the channel width is supposed to be, and what types of switch blocks and connection boxes are to be used for routing. The timing of the device is detailed with specific parameters as well, governing the transmission of the timing signal. VPR is capable to process heterogeneous designs, which means the designer of the architecture may specify more than one type of logic block. Parameters for the logic blocks are for example

the number and position of their inputs and outputs and also the timing parameters for their sequential and combinational input and output. All these components determine the placement and routing process.

The objective of the placement is to assign the functionality of the circuit to the logic blocks available in the FPGA architecture. The algorithm utilized for this task places connected logic blocks as closely together as possible to allow for the routing to be optimal under certain criteria. Such criteria can be a minimal use of routing resources, overall circuit speed or the length of the critical path. A common type of placement strategy that minimizes such criteria uses simulated annealing which minimizes the physical temperature of the routing. VPR realizes simulated annealing with different placement algorithms and different cost functions used for the placement process. The placement algorithm can either minimize the wire length of the circuit or be timing driven and minimize the critical path delay. The different cost functions consider wire length, congestion, and channel width of the FPGA architecture as they try to estimate a future routing. The optimization goals of the placement process can be adapted to the respective FPGA architecture.

The routing process determines how the connections between the now placed logic blocks are realized. Figure 2.6 displays the routing resources available for a circuit placed upon an FPGA architecture. The highlighted routing resources such as wires, switch boxes and connection boxes show the large number of possible routings. The objective of the routing is to find a path through the routing resources that connects the logic blocks in a way that uses minimal resources, is as fast as possible and minimizes congestion. The Pathfinder, see [66], offers a trade-off between delay and congestion. Timing-critical connections, i.e. nets, use a routing that minimizes their delay, non-timing-critical nets use a routing that minimizes the congestion of the resources. VPR uses a Pathfinder-based algorithm that offers different heuristics to achieve a successful routing with a varying degree of focus on minimizing the circuit speed. For this purpose, several cost functions define the cost for the routing by summarizing the amount of used components and delays. The functions let the user define the foci of cost minimization or a trade-off of the different factors. See Figure 2.8 for an overview of a placed and routed design and Figure 2.7 for an extract of the same design displaying configurable logic blocks (clb) and routing resources.

2.2 Security Concepts for Reconfigurable Hardware

2.2.1 Threats and Security Risks

This section reviews approaches to security for reconfigurable hardware. This research area is fast growing and has gained interest only recently, one of the first notions of trust and trustworthiness for reconfigurable hardware have been presented in 2007 by Irvine and Levitt, see [43].

Kastner and Huffmire present an overview of security risks present in the life cycle of reconfigurable hardware in [45] which they divide into three stages: manufacturing, application development, and deployment. During the manufacturing stage, FPGA vendors design and manufacture the hardware. A main challenge is the involvement of different

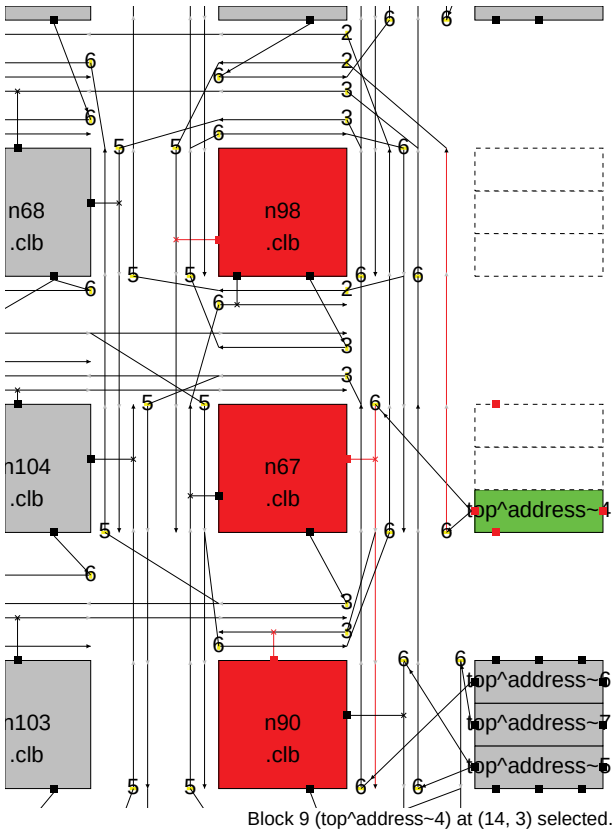


Figure 2.7: Routing detail from from a placed and routed circuit. The global 'topâddress4̃' (green) is routed to three logic blocks (red) using horizontal and vertical routing tracks.

parties and companies and the keeping of trade secrets and design specification. During the application development stage, the future tasks are considered as its programming is done. This stage is determined by design tool subversion, as design and synthesis tools as well as IP cores (hardware modules) of varying sources with different levels of trust are incorporated in the application development. Unwillingly or with full intention, a single tool or IP core used for development might compromise the security of the embedded system. The final stage is the deployment. Depending on the task, devices could be deployed in the most hazardous environment. The vulnerability of such a device depends therefore on its application and the resulting physical environment and as well as the importance of the functionality it performs. The general security risks manifest themselves in specific attacks on reconfigurable hardware possible under the given circumstances.

The involvement of multiple suppliers in the manufacturing stage can lead to the question whether the final product is trusted when design specifications were made accessible to other suppliers. With design specifications known to third parties, a party involved in the production could use that knowledge to integrate additional functionality into the

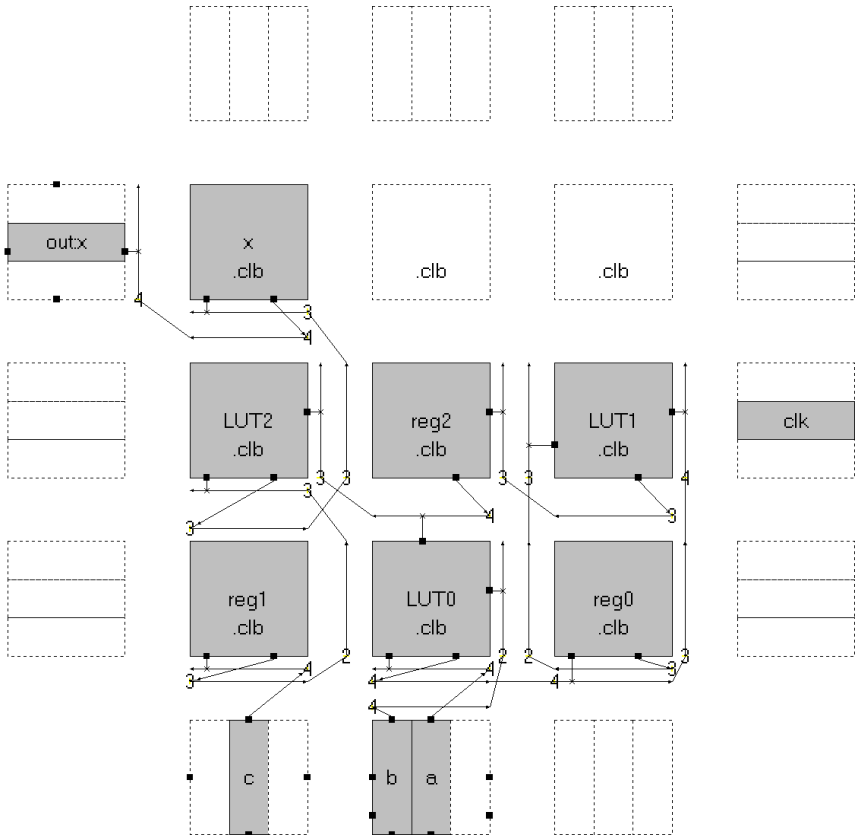


Figure 2.8: Circuit from Figure 2.6 placed and routed by VPR.

hardware. Such a malicious additional functionality hidden in the hardware, called a Trojan horse, could perform potentially damaging action. The activation can occur through any action or system state without the user's knowledge. Variations of the Trojan horse are kill switches and backdoors, rendering the chip inoperable or granting the attacker access to the system, respectively. Karri and Rajendran give a taxonomy and elaboration of hardware Trojans in [44].

Either party that has physical access to the device with or without permission could perform physical attacks. Those attacks are common to obtain sensitive data. The bitstream itself might be a possible target for design theft. Cloning of the bitstream, reverse engineering of the bitstream, and read-back attacks aim to provide an unauthorized party with design information to build a similar device. A device already in use could be exposed to a side channel attack. These attacks make use of the physical behavior of the FPGA and gain information regarding power consumption and thereby gaining information about secret data.

A possible attack on a deployed system is a denial-of-service attack. Such an attack prevents the system from delivering its intended service by destroying the FPGA, changing its programming or shutting it down. An attacker could send too many service requests

or those that the system is not equipped to process. Another possibility to corrupt a deployed system is to intercept the reconfiguration bitstream transmission. This gives an attacker the chance to manipulate the bitstream by changing its functionality or to pass off older bitstreams with known security gaps instead and later exploit the security gaps.

The reconfiguration of programmable hardware allows for design updates and adaption to the environment. Partial reconfiguration gives even greater flexibility to adjust the FPGA's functionality at runtime. Yet, reconfiguration of deployed hardware is a security risk in itself, as it offers many ways of malicious tampering in form of the attacks mentioned as either state in the life cycle of reconfigurable hardware is prone to security risks. At each level in the development, the HDL level, synthesis level, and bitstream level, Mehrhad et al. list possible security threats in [63]. A successful approach to reconfigurable hardware security must therefore validate each step of the life cycle of an reconfigurable device.

2.2.2 Approaches to Hardware Security

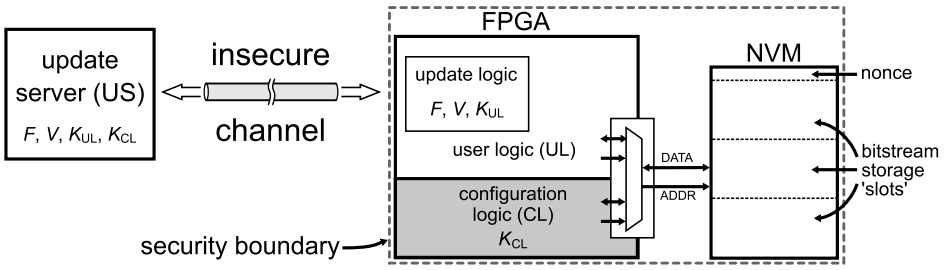


Figure 2.9: Remote Update Scenario by Drimer et al. [26]: An update server (US) installs a new bitstream in a system's non-volatile memory (NVM) over an insecure channel by passing it through update logic in the FPGA's user logic (UL). After reset, the hard-wired configuration logic (CL) loads the new bitstream. For this, the non-secret FPGA identifier F, the version ID V of the operating bitstream, and the secret keys K_{UL}, K_{CL}, and the nonce variable are used.

A first step towards security of dynamically reconfigurable hardware is to establish trust in the bitstream transmission. Typically, FPGA bitstreams are minimally secured by check sums and some FPGA vendors even offer built-in hardware support for bitstream decryption and embedded keys. Chaves et al. [21] propose a more flexible approach based on hashing the bitstream to secure a correct bitstream delivery. Since the bitstream format for the Xilinx Virtex II is packet based, the hash value allows for an attestation of the hardware structure and can be compared to the expected value. This approach makes use of the time necessary for reconfiguration and computes the digest message of the reconfiguration bitstream while loading the bitstream, thus enabling on-the-fly attestation of the bitstream. Since the complete hash value can only be calculated after the delivery of the complete bitstream, region delimitation is enforced in addition to the hardware

attestation. For this, the bitstream is interpreted to determine which physical regions of the device are being rewritten at some specific point in time to avoid memory access outside the intended area.

Drimer and Kuhn [26, 25] distinguish between authentication and confidentiality. They discuss an encryption based security protocol that combines both aspects, authentication and confidentiality, to prevent system downgrades and thereby guarantee the freshness of the reconfiguration bitstream. They suggest the use of either an embedded device ID that is fixed or a device parameter embedded in the bitstream and unique to every bitstream. This approach is aimed at preventing replay attacks where an older version of a circuit is maliciously transmitted to the FPGA to exploit older security vulnerabilities. Figure 2.9 depicts the scenario.

Similar to this, Badrignans et al. propose a combination of special architecture and protocol to avoid the usage of old configurations due to a man-in-the-middle attack in [9] and [24]. They assume that the FPGA to be reconfigured includes non-volatile memory which stores the key used for encryption. It also stores the value TAG_F which specifies the current version of the reconfiguration where only the operator of the FPGA can update it. The remote update process requires two steps, one for initiating the update and one for sending the new bitstream. Each step is initiated by the system designer with a valid new TAG_{UL} which the FPGA compares to its own TAG_F. If the version tags match, the FPGA sends back an acknowledgment key based on the new tag. By mutual updating of the respective tags and their comparison, the protocol avoids a replay attack or an update failure.

The same authors also present a set of countermeasures against physical attacks dedicated to FPGAs in [8], such as masking and hiding to countermeasure side channel attacks in form of power analysis. Masking is based on concealing certain data with the help of a scheme of applying random numbers. A specific value only exists in the form of a masked variable and a function to restore it. Masking results in an averaging of the power consumption since a constant power consumption is the goal of the hiding technique. Each variable consists of two signals that are complementary to each other. Within such a pair, only one signal can switch. This balances the power consumption and hides the variables.

In [36], Huffmire et al. propose to secure IP cores on FPGAs with physical isolation primitives called moats and drawbridges. While moats are unconfigured logic blocks that prevent unwanted and unanticipated communication between cores, drawbridges are selected channels that allow for controlled and secure communication between the cores, see Figure 2.10. The concept of moats and drawbridges is especially useful when combined with a reference monitor, see [40, 39]. For a multi-core reconfigurable system, a reference monitor is installed as an additional core. All remaining IP cores direct their memory access requests to the reference monitor, as it is the only module with direct memory access, see Figure 4.9 in Section 4.3. The monitor grants or denies memory access to other modules, according to an access policy. The access policies are written in a formal language, resulting in either a state-free (not counting the error state or initial state) or a stateful automaton in Verilog HDL. The various policies implement different memory protection schemes. Policy examples are the simple isolation, coordination of

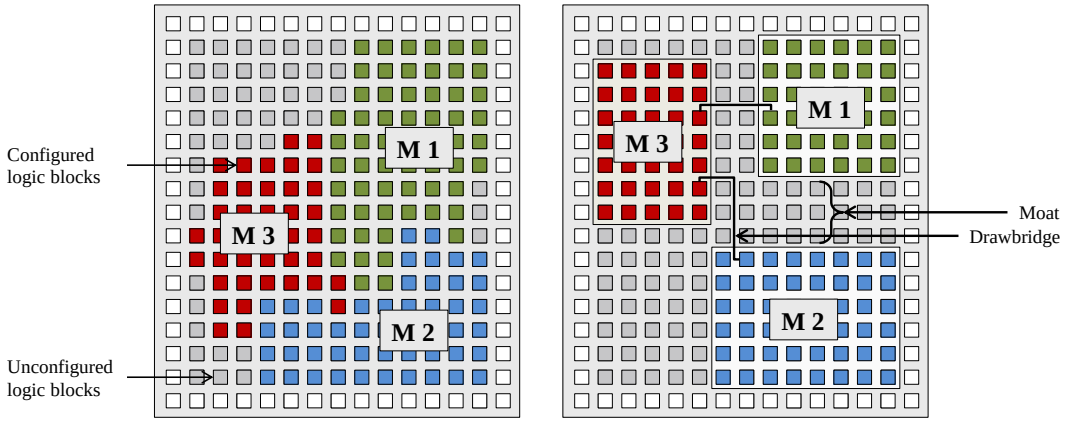


Figure 2.10: Left: FPGA chip with multiple modules placed and routed in the regular, interleaved way. Right: modules are placed in separate areas of the reconfigurable device to enforce physical isolation with only approved routing between the modules.

conflict-of-interest classes, data confidentiality, and data integrity. For further details as well as the combination of physical isolation primitives and reference monitors, see [37] and also [34, 35, 38].

These techniques, however, assume that the module producer can be trusted to implement the correct functionality and do not inspect functional properties of the reconfigurable modules at runtime. Sing and Lilleroth present a core verification flow in [62]. The scenario entails an untrusted hardware core delivered to the consumer who then performs a security check. The core is formally verified by comparing it to its documented or rewritten behavioral HDL code or register transfer-level code. With a bottom up approach, the tool flow decomposes the hardware core into its sub-components and generates according specification netlists and implementation netlists. A formal prover is utilized to analyze whether or not the two circuits deliver identical output under all possible inputs.

Todman and Luk [64] focus in particular on reconfigurable hardware and present a technique to verify the reconfigurable hardware after its compilation, i.e. optimization. The MaxCompiler transforms Java software code into reconfigurable hardware, i.e. reconfigurable streaming design. For this streaming design, a symbolic simulator performs word-level simulation whose output are symbolic expressions. Those expressions are checked for semantic equivalence to ensure that the source and target of the streaming design are semantically equivalent. If the equivalence check succeeds, the software and hardware are trusted to implement the same functionality. This technique could be applied to either hardware or software and has the potential to bring design validation to computer systems that can run a certain functionality in either hardware or software and switch between those two options at runtime. It does not, however, account for other types of verification or validation of software or reconfigurable hardware properties.

It is noticeable how even current approaches to reconfigurable hardware security tend

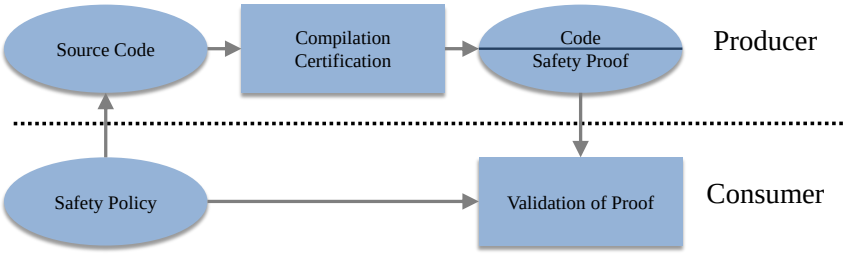


Figure 2.11: Abstract work flow for proof-carrying code.

to focus on one aspect of safe hardware reconfiguration, such as secure transmission or memory protection. With the exception of the approach by Singh and Lilleroth and Anonymous, there is no attempt to understand the behavior of the hardware module. The verification of the behavior of hardware modules is a desirable feature though, as it can attest to more than one aspect of hardware security at the same time and combine otherwise disjoint security properties.

2.3 Proof-Carrying Code

Since it is essential for the understanding of this thesis, I will discuss the proof-carrying code approach to secure software. Necula and Lee introduced Proof-Carrying Code as a mechanism to determine the safety of software from untrusted sources, see [57]. The work flow of the process involves two parties: The first unit, the code consumer, is the target computer system but also its administrator and designer. I refer to the consumer in both functions, a human being with analytical and decision-making skills and a computer system with certain computational capabilities. The second party and counterpart to the code consumer is the code producer. With the term code producer I denote an external software development facility which includes the human code designer as well as the computer equipment used. The code producer functions as an untrusted agent: Upon request, he delivers software code according to predefined security standards set up by the consumer. A formal proof, which is also computed by the code producer, is added to extend the code to proof-carrying code. The code consumer checks the proof of the code against its own safety policy which contains the predefined safety standards. He relies thereby only on the correctness of the proof checker to fully trust the code. Knowing that the code abides the safety policy, the code can be safely executed. The work flow is pictured in Figure 2.11.

2.3.1 Safety Policy

The safety policy is an integral part of proof-carrying code. The consumer defines the desired code behavior within the safety policy and validates the safety proof against it. The safety policy contains a multitude of information which serve as safety rules and calling convention. Authorized operations and their associated safety preconditions make

$$
\begin{array}{lcll}
INSTR & ::= & \text{ADD} & \mathbf{r}_s,op,\mathbf{r}_d \\
 & & \mid \text{SUB} & \mathbf{r}_s,op,\mathbf{r}_d \\
 & & \mid \text{LD} & \mathbf{r}_d,n(\mathbf{r}_s) \\
 & & \mid \text{ST} & \mathbf{r}_s,n(\mathbf{r}_d) \\
 & & \mid \text{BEQ} & \mathbf{r}_s,n \\
 & & \mid \text{BNQ} & \mathbf{r}_s,n \\
 & & \mid \text{RET} & \\
 & & \mid \text{INV} & p \\
 & & & \\
Op & ::= & n \mid \mathbf{r}_i &
\end{array}
$$

Figure 2.12: Subset of the DEC Alpha assembly language with invariant instruction.

up the safety rules, invariants holding when the code to be verified or code native to the host system is evoked define the calling convention. This is sufficient to define safe behavior as low-level or abstract as required by the host system designer (consumer).

Necula and Lee give a practical example of a safety policy for code written in a subset of the DEC Alpha processor language, see Figure 2.12. In this context, $\mathbf{r}_s$ and $\mathbf{r}_d$ denote the source and destination register, respectively.

The safety policy is defined through three items:

- The safety predicate defines the actual safe behavior and is proven to hold true against a set of axioms. In the example given, a language of expressions and memory expression is the basis for specifying expressions that mark addresses which can be safely read or written in context of the state of the program.

- Precondition and postcondition functions as calling convention between the target host system and the proof-carrying code binaries. Through a language of predicates, they indicate the state of the system when the consumer invokes the proof-carrying code as well as when the code calls functions provided by the host system.

- A Floyd-style verification-condition generator (VC generator) which takes the code to be verified and computes a safety predicate in first-order logic.

An abstract machine for memory-safe DEC Alpha Machine Code is created with these components. The abstract machine is a state-transition function that maps a machine state (p, pc) into a new state (p', pc'), where p is the register state and pc is the program counter. The DEC Alpha program is a vector of instructions $\prod$ where the transition from one state to the next is done by executing the current instruction $\prod_i$. The behavior of the abstract machine is described for operations like add, load, store, and branches. As an example, when performing a load instruction, the abstract machine will first check that is is safe to read from the specified address, which has to be aligned by 8.

```
                          %Address of tag in r0
0    INV   Pre_r          %Precondition
1    ADD   r0, 8, r1      %Address of data in r1
2    LD    r0, 8(r0)      %Data in r0
3    LD    r2, -8(r1)     %Tag in r2
4    ADD   r0, 1, r0      %Increment Data in r0
5    BEQ   r2, L1         %Skip if tag == 0
6    INV   r1:addr        %r1 must be readable address
7    ST    r0, 0(r1)      %Write back data
L1   RET                  %DONE
```

Figure 2.13: DEC Alpha assembly code for resource access. The address of the tag is stored in register $\mathbf{r}_0$, the data is stored with an offset of 8 from register $\mathbf{r}_0$. Code lines 0 and 6 contain the annotations with invariants.

2.3.2 Verification Process

I shall now outline how code written in the DEC Alpha processor language can be verified. The code has to be designed to fulfill the security requirements. The code is then annotated with invariants as shown in Figure 2.13. The according postcondition is the Boolean value *true* and is therefore omitted. The function of the code example is to increment the data word if and only if it is writeable. The unnecessary complications within the code are included on purpose to demonstrate that generating and validating safety proofs is still feasible with low-level code transformations. The code therefore contains scheduled instructions and register allocations.

The annotated code and the VC generator let the producer compute the safety predicate. That safety predicate is the actual safety proof which the consumer will check. It is a function of the code annotated with invariants and the precondition and postconditions: For any initial state that satisfies the precondition Inv_0, the code $\prod$ starts executing with the first instruction without blocking; once terminated the final state has to satisfy the postcondition:

$$SP(\prod, Inv, Post) = \forall \mathbf{r}_k \cdot \bigwedge_{i \in Inv} Inv_i \supset VC_{i+1}.$$

The safety predicate for the code example above is the following:

$$\begin{array}{ll} SP_r = \forall \mathbf{r_0}.\forall \mathbf{r_1}.\forall \mathbf{r_m}. & \left.\begin{array}{l} (Pre_r \supset ((\mathbf{r_0} \oplus 8) \ominus 8 : \mathbf{ro\_addr} \wedge \mathbf{r_0} \oplus 8 : \mathbf{ro\_addr}) \wedge \\ (\mathbf{sel}(\mathbf{r_m}, (\mathbf{r_0} \oplus 8) \ominus 8) = 0 \supset \mathbf{true}) \wedge \\ (\mathbf{sel}(\mathbf{r_m}, (\mathbf{r_0} \oplus 8) \ominus 8) \neq 0 \supset \mathbf{r_0} \oplus 8 : \mathbf{addr})) \end{array}\right\} precondition \\ & \left.\begin{array}{l} \wedge\ (\mathbf{r_1} : \mathbf{addr} \supset \mathbf{r_1} : \mathbf{addr}) \end{array}\right\} invariant \end{array}$$

There are two conjuncts which correspond to the precondition and the invariant from line 6 of the example code, see Figure 2.13. The meaning of the first conjunct is that for all values of the registers $\mathbf{r}_0$ and $\mathbf{r}_1$ and every state of the memory $\mathbf{r_m}$ that satisfy

$INSTR$	::=	...	Previously defined instructions
	| SHL	$\mathbf{r}_s$,op,$\mathbf{r}_d$	Shift left $\mathbf{r}_s$ by op bits
	| SHR	$\mathbf{r}_s$,op,$\mathbf{r}_d$	Shift right $\mathbf{r}_s$ by op bits
	| EXTW	$\mathbf{r}_s$,op,$\mathbf{r}_d$	Extract word (2 bytes) from position op in register $\mathbf{r}_s$
	| EXTB	$\mathbf{r}_s$,op,$\mathbf{r}_d$	Extract byte from position op in register $\mathbf{r}_s$
	| LDAH	$\mathbf{r}_d$,n[$\mathbf{r}_s$]	Add $n * 2^{16}$ to $\mathbf{r}_s$

Figure 2.14: Additional instructions to extend the subset given in Figure 2.12.

the precondition Pre_r, the memory locations specified must be readable and in case the tag is non zero, the data address must be writeable. The second conjunct is without any additional information, as is line 6 of the code.

In a final step, the producer proves the safety predicate using the rules of the safety policy. The format of the proof has been determined by the consumer and possibly the producer at an earlier stage. The code consumer validates the safety proof and is then in a position to trust the code. A trusted proof checker as the only requirement enables the code consumer to execute code delivered by an untrusted agent at a cost lower than computing the formal proof.

2.3.3 Potential of Proof-Carrying Code

To explore the potential of proof-carrying code, Necula and Lee present in [57] several case studies that give an indication of how versatile proof-carrying code can be because of its custom made safety policies. One of these case studies is the implementation of a safe packet filter, a proof-carrying code binary whose safety policy includes the four following rules: The memory reads take place only in the area assigned to the package and scratch memory, memory writes take place only in the scratch memory, all branches are forward, reserved and callee-saves registers remain untouched. For a valid package and a valid scratch memory address, this guarantees memory safety and termination. The language of predicates, mentioned as one item that makes up the safety policy, contains a system of types. In the above example, the two only types are addresses that allow for reading and writing and those that allow for reading only. For the safe packet filter, the types now include arrays, a sequence of memory allocations. An array is defined by its starting address, its length, and the type of its elements. Similar to the regular addresses, arrays are designed for memory safety as their memory range always begins on an aligned address and the end of an array keeps a distance to the end of virtual memory to avoid overflow of address arithmetic. Also, two arrays shall never overlap. It is necessary to extend the DEC Alpha Assembly language presented in Figure 2.12 to include shift operations, the extraction of bytes and words (two bytes), and the loading of an effective address of a specified data item (load address high). The abstract machine is redefined, see Figure 2.14, to include these operations in a way that guarantees memory safety. With these few extensions, various types of packet filters can be implemented and verified, hence benefit from formal verification.

Since its introduction in 1996, proof-carrying code has remained an active field of research. In the following works, the potential of the proof-carrying code concept is further elaborated. In 2000, Colby et al. show a proof-carrying code architecture for Java, see [22]: A software-development tool produces annotated x86 binaries from Java code, i.e. .class files. A PCC layer, i.e. tool written in C, checks the proof-carrying code binaries against a safety policy specified in a variant of the Edinburgh Logical Framework (LF) [32]. Necula also presents adjustments to proof-carrying code to increase the scalability of PCC with regard to proof sizes in [56]. The author presents a new hint-based proof representation along with changes to the PCC checker that improve the scalability of PCC. In a next step, Schneck and Necula turn to the matter of removing the trusted code base of the PCC system, see [60]. To make PCC more flexible and to increase security, the code producer provides the safety policies instead of the code consumer. Instead of proving a safety predicate, the code consumer establishes a soundness theorem which the producer must prove. Although that specific framework proves only a part of the safety policy, the soundness of the proof, it clearly demonstrates the flexibility of the proof-carrying concept as it is not bound to a single method of proving. Necula et al. combine the security of PCC and the trust of digital signatures, i.e. a logical framework and an authorization framework, in [65]. This particular combination demonstrates another aspect of the flexibility inherent in PCC: not only can PCC be employed for different verification problems but also different verification scenarios. Some security properties can be proven formally with PCC while others are trusted due to a digital signature. By providing supplemental security guarantees with different ways of authorization, the proof-carrying code principle can be utilized to cover an even wider ranges of security and safety.

The concept of proof-carrying code has been transitioned in other contexts as well. In [48, 47, 46], Klohs and Kastens apply the concept to the validation of program analysis results of software. There are various usages for program analysis results. Such results are for example used for program optimization or the validation of certain software properties, security-relevant or of other type. One application example is the memory safety of Java Bytecode. In accordance to the original proof-carrying code approach, an untrusted producer delivers software code and a proof of certain software properties which is validated against previously defined standards by the consumer of the new software. The objective in this context is to transmit valuable program analysis results which a consumer prefers not to compute himself but still finds necessary to obtain. As with Necula's approach, the original source code is annotated with additional information, in this case interprocedural and intraprocedural summary functions that hold the analysis results. The validation of these results by the code consumer can be achieved with less resources than the program analysis itself. Hence, this approach is applicable to scenarios where the consumer receives software from an untrusted producer and the parties have different computational resources. In some cases, the computational resources of the consumer, i.e. the target platform, can even be considerably smaller than the producer's: Klohs and Kastens apply in [48] the proof-carrying code technique to the verification of Java Bytecode on limited devices, e.g. smart cards. The verification of Bytecode guarantees that every operation always operates on objects of the correct type. The objective of this scenario is to provide

the smart card with information to perform the Bytecode verification in a memory-optimal manner, i.e. to optimize the performance of the verification. With the program analysis information available, the memory footprint for the verification could be dramatically reduced.

The proof-carrying hardware intellectual property (PCHIP) by Love et al., see [51, 50], is an approach similar in name to the approach featured in this work but published after my initial publications. PCHIP is an approach for formal validation of security-related properties of hardware modules. It is focused on the functional validation of an IP core's behavior: An IP vendor sells hardware modules at register-transfer level (RTL). The specific safe behavior of the IP core at RTL level is agreed upon between the vendor and the consumer and modeled in a subset of Verilog [51] with a novel set of definitions in the Coq formal language [5]. With the Coq framework, the consumer validates the proof of correct behavior of the IP core. This approach differs from my work as it is concerned with the security-relevant behavior of hardware at the RTL level and only security-relevant hardware properties, described in the Coq framework. The approach by Love et al. is one possible instance of the general proof-carrying hardware concept. To the best of my knowledge, the concept of proof-carrying hardware as described in this thesis is a novel concept that has never before been developed.

2.4 Chapter Conclusion

This chapter gave the background information and the context for this thesis. I discussed FPGAs as programmable hardware, related work in the area of hardware security and elaborated on proof-carrying code as a basis for a new approach to hardware security.

The review of related work in Section 2.2 gave an overview of the current works in the research field of reconfigurable hardware security. Existing security concepts generally deal with one aspect of safety, such as memory safety or secure transmission, but are not extendable to other aspects of security. An approach to security that is adaptable to various safety features of reconfigurable hardware while allowing for fast validation of safety proofs and therefore quick reconfiguration with a minimum workload for the FPGA is missing.

The following chapter discusses key concepts and ideas of proof-carrying hardware, a safety concept for reconfigurable hardware based on the principles of proof-carrying code as elaborated in Section 2.3.

CHAPTER 3

Key Concepts and Ideas

This chapter describes the context as well as key concepts and ideas of the novel proof-carrying hardware approach to security for reconfigurable hardware devices. In Section 3.1, I give an overview of work in the fields of software and hardware security related to this thesis. In Section 3.3, I elaborate on the origin of the proof-carrying hardware principle and the key challenges for a novel security concept for reconfigurable hardware that is distinctly different to other concepts of that domain, as discussed in Section 3.4. Section 3.5 states the methodology for this project and claim of this thesis.

3.1 Software Security and Hardware Security

In this section, I outline the main challenges for developing a novel security concept for reconfigurable hardware. I assume a scenario as depicted in Figure 3.1 where new hardware modules are delivered by an untrusted source through an unsecured channel to the reconfigurable platform. Chapter 2 gave an introduction to proof-carrying code. Since proof-carrying code is a security concept for the software domain, it needs to be transferred to the domain of hardware security, in particular the domain of reconfigurable hardware security. Hence, the differences between software security and hardware security need to be considered. Secondly, the scenario in which FPGAs as embedded systems are to be secured needs to be taken into account when creating a novel security concept for reconfigurable hardware.

There are several characteristics of the software security domain that do not apply to the domain of reconfigurable hardware security:

- Software is compiled for an instruction set architecture (ISA) and deployed into an established software-ecosystem. As a result, it is known which instructions are performed, how they work, and what reactions may result within the environment. Reconfigurable hardware consists of a large number of spatially arranged (placed

and routed) components which can form a new and non-standardized execution environment. This bears the question what system reaction an instruction might trigger.

- The research for safety and security is rather advanced for the software domain as challenges and concepts are well-known. For the reconfigurable hardware domain, that particular field of research has only been emerging for a few years. As a result, concepts are new and often focused on one aspect, see Section 2.2. What is missing is a concept for hardware security that is flexible enough to accommodate any hardware configuration and also changes of the configuration.

- Security for reconfigurable hardware poses a particular challenge due to its specific scenario. Formal or other elaborate verification of incoming bitstreams on the spot is usually not feasible for reconfigurable platforms as they have limited available resources. The reason for those limitations vary, some platforms may not have the necessary computational power to do the elaborate computation of a formal proof while other platforms occupy their resources otherwise. Even if the reconfigurable system could provide the necessary resources, such as data compute centers with large capacities, it may not always be economical to employ those resources. In certain reconfiguration scenarios though, a minimized reconfiguration time is necessary. This requires to assess as quickly as possible whether a new module can be trusted. A successful approach to reconfigurable hardware security should allow for an on-the-fly or otherwise fast enough validation of security features that can actually be performed by reconfigurable systems with the above limitations.

With the transition of proof-carrying code to proof-carrying hardware, an approach to hardware security is delivered, that takes on these challenges:

- In Section 2.3, I elaborate the annotation of assembler code performed by proof-carrying code. In a language of predicates, preconditions and postconditions describe the state in which the system should be at the given point in the program. A verification-condition generator takes the annotated code and computes a safety predicate that proves the safe behavior according to previously established standards, i.e. languages of expressions and memory expressions.
  There are two aspects to this concept: The annotation of the source code and the computation of a proof to demonstrate the adherence to certain criteria. Both aspects could, in some form, be applied to reconfigurable hardware. As mentioned before, reconfigurable hardware is realized by a circuit which is placed and routed on the reconfigurable (programmable) fabric. The process, detailed in Section 2.1, begins with a circuit design in a hardware description language and undergoes several transformations until it becomes a placed and routed netlist. Depending on the desired form in which the hardware module is transmitted from a producer to a consumer, different types of annotations could be considered. A netlist may be annotated to indicate the presence of certain elements that are wanted. If the circuit

is represented as a graph, specific structures within the graph may be highlighted as they result in desirable features of the placed and routed circuit resulting from that graph. Note that a desirable feature could range from security assurances, on which this work focuses, but could also regard other non-security critical features. With or without annotation of the hardware module, proof-carrying hardware can be used to convey a proof whose type and degree of formality suits the aim of the verification.

- I elaborated that proof-carrying hardware is an approach to verify properties of reconfigurable hardware and is not limited to properties that are relevant to security and safety questions. As I shall explain in Section 3.3, proof-carrying hardware is a flexible concept. It can therefore be applied to already existing approaches to hardware security and makes use of the already existing research in the field of security for reconfigurable hardware.

- Proof-carrying hardware is particularly suitable for an application in scenarios that feature a reconfigurable target system with limited resources for verification. As in the case of proof-carrying code, the producer of the hardware module invests the resources to compute the proof that the module adheres to specified standards. The reconfigurable system is merely burdened with the validation of the proof. Proof-carrying hardware also meets this challenge by its flexibility to adapt the type and format of the proof to suit each individual scenario and reconfigurable system.

The methodical approach of this work is the transition of proof-carrying code to the domain of reconfigurable hardware. By meeting the above challenges, proof-carrying hardware is a usable, flexible, and robust approach to verify, formally or otherwise, properties of the hardware modules for reconfigurable computer systems. Those properties may be functional or non-functional, i.e. deal with the functionality of the hardware module or the physical aspects, and could cover a wide range of aspects of reconfigurable hardware, e.g. security, performance, or other attributes of the module's composition.

3.2 Methodology: Application of the Proof-Carrying Code Principle

I introduced in Section 2.3 the concept of proof-carrying code: The consumer of software requires that a certain property holds at certain or all points of program execution. The producer constructs a proof of that property for the program, e.g. in Hoare Logic. The proof is computed from the source code which the producer annotated with pre- and post-conditions (assertions) based on a language that formulates certain (safe) behavior. The annotated code and the proof are sent to the consumer. The consumer checks the proof for correctness and whether it guarantees to desired properties. The consumer also utilizes the code annotations to reconstruct the proof, check the proof, and match the proof to the software code. Klohs and Kastens applied the concept of proof-carrying code to a

different verification problem and scenario: The consumer, a Java Virtual Machine on a smartcard, needs to apply program analysis on received Java Bytecode for two reasons; to check for type safety and to perform program optimization. The producer annotates the the Java Bytecode with interprocedural and intraprocedural summary functions that hold the analysis results. The validation of these results by the code consumer can be achieved with less resources than the program analysis itself.

The proof-carrying code concept can be regarded in a much broader sense: any software or hardware can be complemented by additional information that are more expensive to compute than to validate. The type of those information is not limited to those attesting to security or safety aspects of the delivered code. The methodical approach of this work is the adoption of the concept inherent in proof-carrying code and its application to the domain of hardware (and hardware security in particular) in the form of proof-carrying hardware.

In the proof-carrying hardware scenario implemented for this thesis, a consumer requires equivalence between his design specification and the implemented hardware module. A producer creates a hardware module according to the specification file which outlines the entire circuit. The implementation and specification of the circuit are combined in a miter function in cnf form which is unsatisfiable if and only if the two circuits are equivalent. The producer computes an elaborate resolution proof of the miter's unsatisfiability. The consumer receives the complete hardware module in bitstream format and the proof. He validates this proof and the logic extracted from the bitstream against the complete design specifications. While this work focuses on the particular security aspects of design and specification equivalence of reconfigurable hardware, the concept of proof-carrying hardware is not limited to that domain. Proof-carrying hardware has the potential to become a container for information regarding various aspects of reconfigurable hardware. I shall list four possible further applications of proof-carrying hardware:

A consumer orders a hardware functionality represented as a graph. To ensure the existence of certain properties, the graph has to include specific patterns. The producer who composes the graph utilizes computationally elaborate pattern matching to highlight patterns within the graph, such as a full n-bit adder or simply a part of the circuit which has n inputs and m outputs. The entire hardware graph with its annotations that mark the desired patterns is sent to the consumer who quickly validates the existence (and possibly the correctness) of the components.

A consumer requires physical isolation for multiple hardware modules placed and routed on a single FPGA fabric. A producer delivers a bitstream that contains multiple circuits, i.e. hardware modules, for which the placement and routing is performed according to the physical isolation principle called moats and drawbridges introduced by Kastner et al., see [36]. The producer annotates the placement information to indicate which areas are assigned to which circuit and which areas are designated moats, i.e. remain unconfigured. In addition, the moats are proven to be sufficiently large considering the longest routing resources reaching into the moat from the two adjacent hardware modules. The proof is a simple comparison of the moat and the length of the combined routing wires, hence easy to validate by the consumer. The similarity to the approach featuring the pattern

matching for the graph lies in the simple annotation of the respective hardware. A pointer to the according part of the hardware is used for both; the proof of sufficient moat width could be omitted by the producer as the consumer could simply add the numbers himself.

A consumer demands hardware at register-transfer level (RTL) for which certain properties are always guaranteed. A producer designs the RTL hardware module and annotates the sequences of linear code (i.e. code without jumps and loops) between register accesses with preconditions and postconditions in a language similar in principle to those constructed for proof-carrying code. The annotated code is compiled into a formal proof which the consumer checks and matches it based on the annotations against the code. The hardware is proven to hold the desired properties. This approach would be a more direct transition of the original proof-carrying code into the domain of RTL hardware.

A consumer wishes to employ program analysis to obtain hardware that is governed only by specified dependencies. Similar to the approach by Klohs and Kastens, program analysis is performed for security (unwanted dependencies due to faulty construction) and optimization purposes. The producer of the hardware annotates the code with intermediate analysis results. A proof of the existence of only approved dependencies along with the annotated code is sent to the consumer. The consumer is able to quickly validate the proof and match it, with the aid of the annotations, to the code. He receives a diagram of dependencies that has to match the specifications. The difference to the approach by Klohs and Kastens, besides differences in the implementation and formats, lies in the focus on dependencies. For this approach, it is of interest which procedures and variables influence other variable or procedure, but the actual variable value may be only of secondary concern.

There are existing approaches to bring verification to Verilog hardware modules. As mentioned in Section 2.3.3, Love et al. have introduced the concept of proof-carrying hardware intellectual property. A consumer requires hardware that fulfills its functionality in a specific manner, i.e. with secure behavior. A producer composes a hardware module and translates the code into a theorem proving language. Then, the producer computes a formal proof of the desired safe behavior. The consumer is given the annotated code and proof and with the help of the annotations quickly validates the proof against the previously determined security standards and the code. This approach can be viewed as an instance of the general proof-carrying hardware concept. To the best of my knowledge, the concept of proof-carrying hardware is a novel concept that has never before been developed and predates the concept of proof-carrying hardware intellectual property. Proof-carrying hardware is the first approach to bring the concept and full potential of proof-carrying code to the hardware domain. As stated above, proof-carrying hardware is not limited to the verification of security properties of reconfigurable hardware but is instead a method of delivering and validating computationally expensive information to aid the consumer and which can regard a multitude of properties relevant to reconfigurable hardware.

The possibilities proof-carrying hardware as presented with this work are numerous and can potentially cover different forms of hardware as well as different verification challenges.

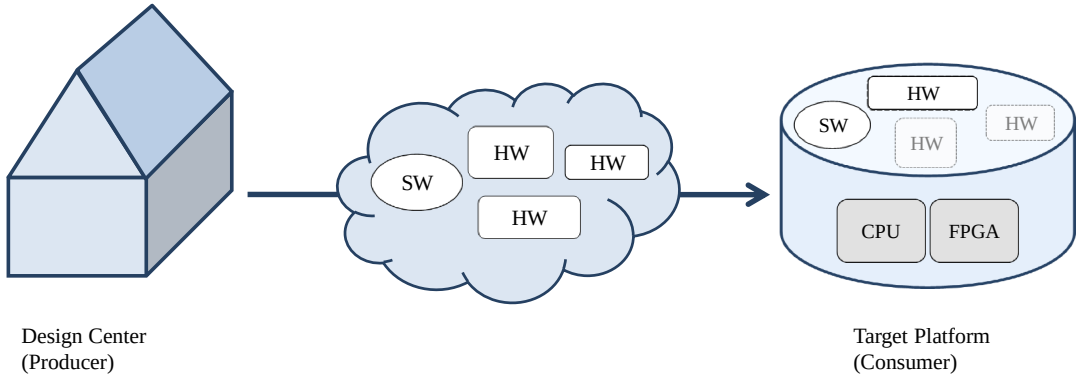


Figure 3.1: Producer-consumer scenario depicting the unsecured and untrusted delivery of hardware modules for a reconfigurable embedded device.

3.3 The Proof-Carrying Hardware Approach

In this section, the integral aspects of the proof-carrying code concept that are fundamental for the development of proof-carrying hardware are identified. These aspects are also integral aspects of proof-carrying hardware, as it is modeled along those. All these aspects center around the split of the work flow between the producer party and consumer party and their respective role and tasks.

In Section 2.3 I explained how proof-carrying code utilizes a safety predicate to prove the code's compliance to previously established standards. Those standards are defined in a set of axioms which define the safe code behavior. The axioms are formulated based on a language for memory expression and a language of expressions, see [57] for details. Necula and Lee present those languages but do not specifically request either the consumer or producer to be responsible for the development of those. For the purpose of this work and proof-carrying hardware, I assume that it is the consumer who defines the safety policy according to the system he is operating and according to his understanding of security. To define the safety policy gives the consumer the freedom to determine what constitutes secure behavior of the hardware module as well as the type of proof and proof validation process. Of course, it is possible for the consumer to include the producer in those decisions or the development of languages to formalize safety rules, if desired. In real life environments, a cooperation between consumer and producer may be very sensible and of great benefit to both parties. In any case, the consumer is knowledgeable about what he expects from the new hardware module with regard to security before the production begins. The safety policy for the proof-carrying hardware prototype presented in this work will be elaborated in Chapter 4.

For both, proof-carrying code and proof-carrying hardware, the following holds true: The hardware module producer caters to that demand for security (and the demand for new hardware functionality, of course). It is the task of the producer to create hardware

that is secure to the previously established standards inherent in the safety policy. It is also the task of the producer to compute the safety proof for verifying the new module's adherence to the safety policy. Hence, the producer is burdened with two major tasks that require computational resources and time.

As a result, the consumer of the new hardware module is left with the task of validating the proof. This is a lightweight task in comparison to the computation of a formal or otherwise extensive proof, especially when considering the tremendous security gain. The validation includes checking the proof's correctness and matching it to the safety policy as well as the bitstream, i.e. hardware module. As the proof is unique to the hardware module and safety policy, it cannot be exchanged for a different (correct) proof that proves a different safety policy. For an overview of the work flow see Figure 3.2. In the case of proof-carrying code, this means for the producer to annotate the assembler code and to compute a safety predicate and for the consumer to validate that safety predicate with a formal proof checker against the safety policy. As stated above, proof-carrying hardware can make use of both aspects, the attachment of a proof or other additional information the the hardware module as well as the annotation of the hardware module. For this work, I shall focus on the extension of a bitstream to a proof-carrying bitstream by adding a proof, see Chapter 4.

In the context of this work, verification and validation denote the above meanings: Verification refers to the process of producing a proof or otherwise providing evidence. Validation is the task of checking a proof in order to assert its correctness. Therefore, the verification is always performed by the producer of a hardware, the validation by the consumer.

Considering the scenario of hardware security for embedded devices, the separation between consumer and producer is vital for the concept proof-carrying hardware. As an approach to hardware security based on the separation of a knowledgeable consumer with requirements and an untrusted but resourceful producer, proof-carrying hardware can meet the demands made for a new concept for reconfigurable hardware security in Section 3.1.

3.3.1 Pillars of Proof-Carrying Hardware

I now outline the key characteristics of the proof-carrying hardware concept:

- Usability of proof-carrying hardware: By shifting the workload to an external source that produces the hardware module and its formal safety proof, proof-carrying hardware offers the safety of formal verification for reconfigurable platforms which can come with the limitations mentioned before. As the mere validation of a formal proof is less costly than the elaborate computation of such, the target platform only has to use little computational resources and time compared to the producer. By providing the smallest workload possible for the consumer, proof-carrying hardware provides the means for instant security combined with on-the-fly reconfiguration, that is a reconfiguration without delay but after a successful validation of the proof of security.

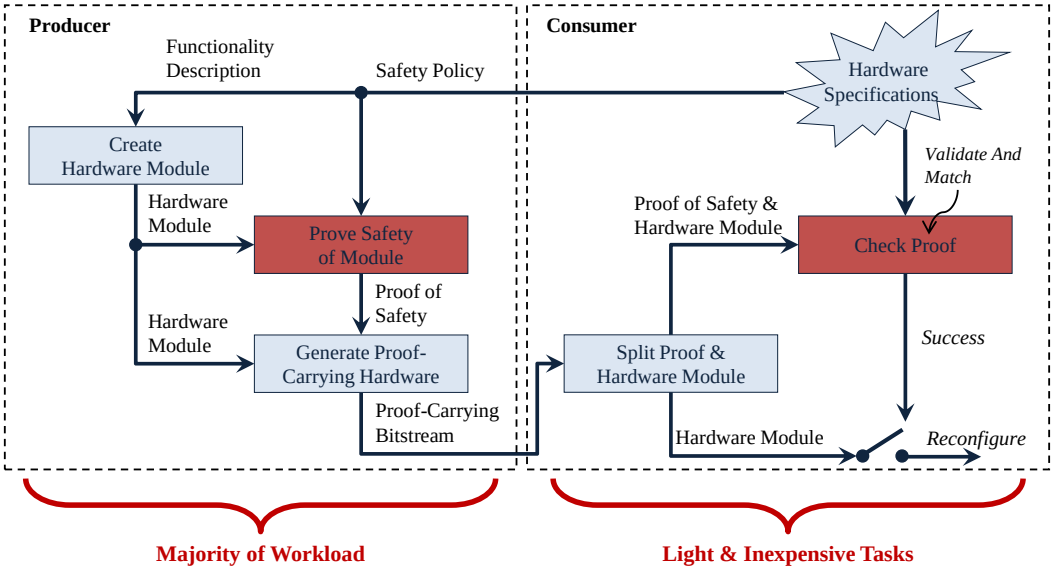


Figure 3.2: The general proof-carrying hardware scenario: The consumer makes demands regarding functionality and safety,the producer performs the majority of the workload by creating the hardware module and proving its safety. The consumer is left only with the lightweight task of checking the safety proof at runtime before reconfiguration.

- Flexibility of proof-carrying hardware: My novel approach offers a more complete understanding of security than techniques that focus on one aspect of reconfigurable systems. Unlike other approaches, proof-carrying hardware performs the verification based on a safety policy. That safety policy is established by the consumer, i.e. the reconfigurable platform host, and can potentially incorporate a multitude of hardware module features that can be formally described and in some way verified. Those features can be extended to vary from software-like input / output range checks regarding the actual functionality of the module to physical characteristics of the hardware realizing the module. In short, the consumer decides on functionality, security features and aspects to be proven, and formats. It is this flexibility that makes proof-carrying hardware capable of handling the versatility of the hardware platforms, as it can be applied to basically any kind of formal proof or verifiable data.

- Robustness of proof-carrying hardware: Proof-carrying hardware guarantees trust in the hardware module without any previous trust in the production or delivery process of the hardware module or any party involved. Any discrepancy between design and design specifications, accident or attack up until the delivery of the final proof-carrying hardware must result in a failure to establish trust in the new hardware module. The flexible safety policy incorporates the consumer's reconfigurable

platform characteristics and its inherent security aspects as well as the desired functionality of the new module. I assume this safety policy to fully reflect all security restrictions and needs of the consumer's platform in the sense that it is complete. If the proof is correct and matches the safety policy as well as the hardware module, it is guaranteed that the hardware module has the specified properties. If either the part of the proof-carrying bitstream, proof or bitstream, was accidentally damaged or maliciously altered, a security check would fail. The proof would either be rendered incorrect or it would not match the hardware and safety policy any longer. This is also an insurance against a situation where producer and consumer operate with different safety policies. The only requirement for the host system is to use a trusted proof checker. This is an acceptable request since the type of proof and proof checker itself will likely not alter as often as new hardware functionalities are needed and it can therefore be regarded as a one-time investment.

3.4 Comparing Proof-Carrying Hardware to Existing Approaches

In Section 2.2, I gave an overview of existing approaches to security for reconfigurable hardware. This section will classify the approaches and compare them to the novel proof-carrying hardware concept.

These first three approaches introduced in Section 2.2, Chaves et al. [21], Badrignans et al. [24], and Drimer and Kuhn [26, 25], aim at detecting and refusing unauthorized bitstreams and to protect the bitstream's confidentiality as well by applying cryptography. Proof-carrying hardware does not rely on a secured transmission to implement the concept of authorized and up-to date bitstreams. The user updates the safety policy for his host system that is to receive the new bitstream. If the new hardware module together with the new proof of security does not adhere to the updated safety policy, the security check will fail. If an adversary successfully passes off an older bitstream that does not fail the security check, which is unlikely but not impossible, it only means that the older version of the bitstream already incorporated the new safety policy. The attacker can therefore not exploit any security weaknesses as they never existed. As for the matter of confidentiality, proof-carrying hardware can be applied to an encrypted and then decrypted bitstream as well. The confidentiality and authenticity of the bitstream are therefore guaranteed with proof-carrying hardware.

Badrignans et al. also emphasize on the importance of preventing side channel attacks and deliver an approach to masking the power consumption. Proof-carrying hardware is designed to ensure a secure reconfiguration of an FPGA. Physical attacks, e.g. read-back attacks, are outside the scope of proof-carrying hardware. Nevertheless, my approach does not interfere with the one presented by Badrignans et al. as they can be applied in combination. In that regard, the approach to masking the power consumption in order to prevent a side channel attack is exemplary for other approaches to aspects of reconfigurable hardware security, or general hardware security, that are positioned outside the scope of proof-carrying hardware.

The moats and drawbridges approach as well as the reference monitor by Huffmire et

al. are examples of security concepts that could and will, as I shall demonstrate for the reference monitors in a later chapter, benefit from proof-carrying hardware. When the placing and routing is performed including moats and drawbridges in the floor planning, the physical isolation and therefore integrity of the memory is achieved. Yet, what is missing is a way for the consumer of a placed and routed hardware module to validate the isolation. The principle of proof-carrying hardware could be applied to the concept of moats and drawbridges to deliver a process of verification and validation of the physical isolation of IP cores on an FPGA. The same holds true for the reference monitor. When a reference monitor that functions according to specification is integrated into the FPGA, the memory access is regulated and can be regarded as secure. But without a feasible validation of some proof of correctness, the user of the reference monitor can never establish trust in the functionality of the reference monitor. In Chapter 4, I will demonstrate how proof-carrying hardware can be utilized to formally verify the equivalence of the specification and implementation of a reference monitor.

The last approach presented in Section 2.2 is the one by Singh and Lilleroth [62] which is concerned with the actual functionality of a hardware module. The consumer decomposes the delivered module and analyzes how it compares to its design specifications, i.e. whether design and specification are equivalent. First of all one notices that it is the consumer and his computer system performing the work of decomposing, analyzing and proving. In following chapters I shall demonstrate the prototype of proof-carrying hardware that gives a formal resolution proof stating the combinational and bounded sequential equivalence of a hardware core to its design specification. And while giving a very similar type and quantity of security, proof-carrying hardware also relieves the consumer of the burden of producing a formal proof by shifting this task to the producer.

This comparison shows that proof-carrying hardware can establish the security given by other approaches or even increases it by adding a proof of security to otherwise unattested security measures. It is also a unique trait of proof-carrying hardware to burden the hardware producer and not the consumer with the task of providing security. For security concepts that target aspects outside of the scope of proof-carrying hardware like physical attacks, my novel concept can still be used in addition to a different approach without having one interfere with the other.

3.5 Thesis Claim

The objective of my project is the development of proof-carrying hardware as a novel security concept for the verification of reconfigurable hardware. The thesis claim is that the following statements are true:

- The transfer of the proof-carrying code concept to the domain of dynamically reconfigurable hardware is feasible.
- The method of Proof-carrying hardware is usable.
- The method of Proof-carrying hardware is robust.

- The method of proof-carrying hardware is flexible.

The development of the proof-carrying hardware concept will be accompanied with a prototype implementation to evaluate the usability, robustness, and flexibility. The usability of proof-carrying hardware will be documented in the shift of workload from consumer to the producer, measured in runtime and memory usage for the producer and consumer. The robustness of proof-carrying hardware will be tested with the prototype implementation that has to detect any tampering with the proof-carrying bitstream. The flexibility of the concept will be established with the application of proof-carrying hardware to selected example scenarios.

3.6 Chapter Conclusion

Proof-carrying hardware is a novel concept for the security of reconfigurable hardware that provides the usability, flexibility, and robustness to achieve runtime verification of reconfiguration bitstream for embedded systems deployed in security critical environments. The key concept is a separation of the work flow between the producer of the hardware module, an untrusted source, and the consumer. The consumer of proof-carrying code specifies his security demands, leaves the production and computation of a safety proof to the producer, and upon delivery performs an on-the-fly validation of the proof to establish trust in the new hardware module. This security concept can be applied to existing security challenges and extended as needed. The task of this project is to design the novel proof-carrying hardware concept, to develop a prototype and to demonstrate how it realizes runtime verification for reconfigurable platforms such as FPGAs.

CHAPTER 4

Proof-Carrying Hardware Approach to Runtime Verification

The verification of a reconfiguration bitstream can investigate a multitude of properties of the delivered circuit. Those properties can either regard the functionality of the circuit or the physical design of the circuit. Both, functional and non-functional properties can be of interest for security verification. Non-functional properties that could be potentially security critical are for example the adherence to a fixed clock frequency, routing constraints, or the physical isolation of multiple IP cores on a single fabric as mentioned in [36]. This work focuses on the equivalence of design specification and circuit implementation, a functional property. The combinational or sequential equivalence eliminates a large variety of security threats, such as hidden functionality as described in Section 2.2.1. It is therefore reasonable to choose these properties for a first application of proof-carrying hardware.

This chapter describes three security challenges for reconfigurable hardware to which I later apply proof-carrying hardware. The three security challenges are the combinational equivalence of design specification and implementation of a combinational circuit, elaborated in Section 4.1, the bounded sequential equivalence of design specification and implementation of a sequential circuit, discussed in Section 4.2, and the verification of combinational and sequential reference monitor hardware modules, see Section 4.3, which regulate the access to external memory attached to an FPGA chip.

4.1 Combinational Equivalence Checks

This section details the verification and validation of combinational equivalence of two circuits, i.e. the specification and implementation of a design, as security feature of reconfigurable hardware to be guaranteed with proof-carrying hardware. The security gain of combinational equivalence checks is substantial as it prevents any alteration of the original functionality description done by the consumer who hosts the reconfigurable target platform. To prevent any alterations of the original design specifications means to prevent numerous security risks, e.g. faulty functionality, but also malicious design additions, such

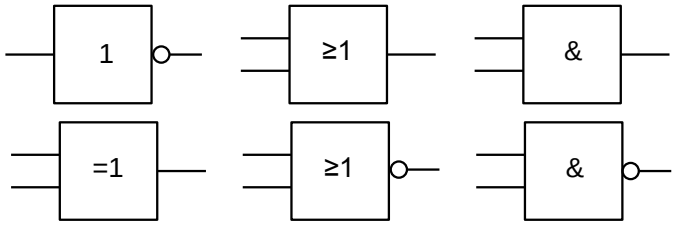


Figure 4.1: The first three gate types are from left to right the NOT gate, OR gate and AND gate. The second row of gates displays the XOR gate, NOR gate, and NAND gate, displayed in the IEC (International Electrotechnical Commission) representation.

X	Y	$\overline{Y}$	X & Y	X OR Y	X XOR Y	X NOR Y	X NAND Y
0	0	1	0	0	0	1	1
0	1	0	0	1	1	0	1
1	0		0	1	1	0	1
1	1		1	1	0	0	0

Table 4.1: Truth table for the three basic NOT, AND, and OR gates as well as the XOR, NOR, and NAND gate.

as hardware Trojans as described in Section 2.2.1.

4.1.1 Combinational Circuits

For the sake of completeness, the following section reviews the fundamental concepts of gates and combinational circuits as it is fundamental in understanding the concept of combinational equivalence.

A combinational circuit is composed of gates that compute the circuit's output as a function of its current Boolean input. The history of the input is irrelevant, the circuit operates without a clock. Each gate has one or more input values and exactly one output value. The three most basic gates are the unary NOT gate, the binary AND gate, and the binary OR gate, see Table 4.1. These three types of gates are sufficient to realize any Boolean function, but can also be implemented using different types of gates such as the XOR gate, the NOR gate, and the NAND gates. Figure 4.1 displays the symbols for the gate types, Table 4.1 the according functionality. The binary gates can be extended to have more than two inputs by concatenating the inputs with the according operation.

The following example outlines the concept of combinational equivalence of two circuits: A half adder is a combinational circuit which calculates the sum S=(X & Y) and carry bit C= ($\overline{X}$ & Y) OR (X & $\overline{Y}$) for the addition of two bits. The first half adder in Figure 4.2 is composed of one OR gate and three AND gates, two of them using negated inputs. The second circuit uses a single XOR gate instead of two AND gates and one NOT gate

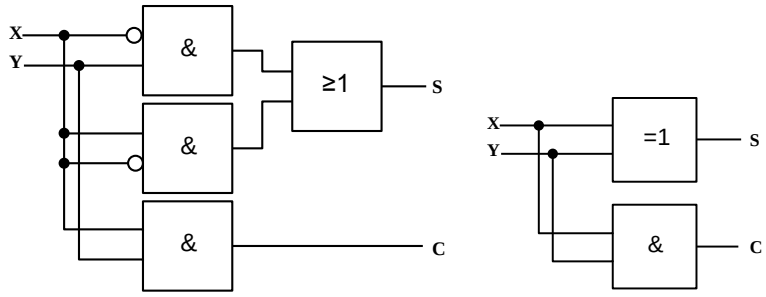


Figure 4.2: Both circuits implement a half adder, i.e. the same function.

X	Y	S=($\overline{X}$ & Y) OR (X & $\overline{Y}$)	S = X XOR Y	C = X & Y
0	0	(1 & 0) OR (0 & 1) = 0	0	0
0	1	(1 & 1) OR (0 & 0) = 1	1	0
1	0	(0 & 0) OR (1 & 1) = 1	1	0
1	1	(0 & 1) OR (1 & 0) = 0	0	1

Table 4.2: Truth table for the two half adder circuits of Figure 4.2. As can be seen in column two and three, the two different realizations of S deliver the same output.

to compute the sum. Both circuits implement the same functionality, as demonstrated by their respective truth tables, see Table 4.2. Other realizations are also possible, for instance an implementation using only NAND gates as any Boolean function can be realized using only NAND gates. Hence, for a combinational circuit, there may be multiple implementations.

4.1.2 Combinational Miter

As elaborated above, there can be more than one circuit for a single Boolean function if the choice of gates and number of gates used for the circuit is not restricted. There can even be multiple circuits where each circuit uses the same number and type of gates that implement the same functionality, i.e. that are combinational equivalent.

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 by a logic function $I(\underline{x})$. 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 combined in a single OR gate to form $M(S(\underline{x}), I(\underline{x}))$. Figure 4.3 shows the construction of the miter. Apparently, the respective number of inputs and outputs of the specification and the implementation must match.

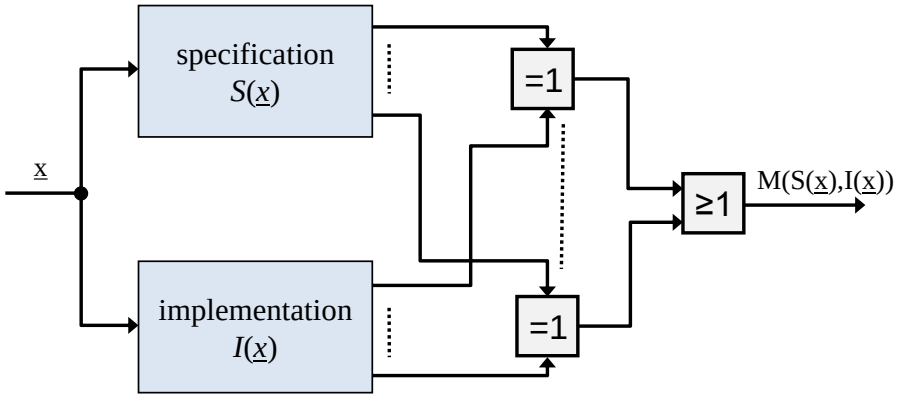


Figure 4.3: Construction of the miter function $M(S(\underline{x}), I(\underline{x}))$ for two representations of a combinational circuit.

The miter is then computed into an Boolean formula to be proven unsatisfiable. A Boolean formula consists of literals, their atomic units which cannot be further divided and usually denote variables, and clauses that are disjunctions of literals, i.e. literals connected with a logical OR. A Boolean formula takes conjunctive normal form (cnf) when it is a conjunction of clauses. A simple example of a Boolean formula in cnf is $(x_1 + \overline{x}_2 + \overline{x}_3) \cdot (x_1 + x_3) \cdot (\overline{x}_1) \cdot (x_2 + \overline{x}_3)$, where an OR and AND are represented with the mathematical symbols for addition and multiplication respectively to improve readability. If under any input $\underline{x}$ the specification and the implementation of a circuit generate different outputs, the miter function will evaluate to 1. Consequently, demonstrating equivalence means to prove the unsatisfiability of the miter, i.e. it can be proven that there exists no input $\underline{x}$ for which the miter function would evaluate to Boolean TRUE. If there is any input under which the miter function is satisfiable, then this input is the proof that there is an input to the two circuits under which those produce different outputs. Modern CEC tools like ABC [4] internally represent the miter with And-Invertor Graphs (AIG), rely on Boolean satisfiability (SAT) solvers and output the result in conjunctive normal form in DIMACS format [3].

4.1.3 Verification and Validation with Resolution Proofs

During the last years, SAT solvers have progressed into tools that can generate resolution proofs for unsatisfiability. The improvement of such techniques are still a focus of research such as [54, 20, 55, 18] and also [31]. A resolution proof is a sequence of resolutions of the original cnf and intermediate clauses that eventually results in an empty clause which models a contradiction. Hence, a formula is unsatisfiable if a resolution proof exists. Otherwise, an example of values for the literals can be found that makes the formula, in this context the miter function, evaluate to TRUE.

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, Table 4.3 gives a possible proof

trace for the cnf $(x_1 + \overline{x}_2 + \overline{x}_3) \cdot (x_1 + x_3) \cdot (\overline{x}_1) \cdot (x_2 + \overline{x}_3)$: 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. In each step, the literal that is used in its negated form and in its non-negated form can be resolved. The first resolution step utilizes lines 2 and 3 and resolves $x_1 + x_3$ against $\overline{x}_1$, leaving only x_3 as line 5. Line 4, $x_2 + \overline{x}_3$, is then resolved against the new line 5, leaving x_2 as line 6. As a last step, four lines are resolved at once: Lines 3, 5, and 6 contain combined the negated literal present in line 1 which is $x_1 + \overline{x}_2 + \overline{x}_3$ resolved against $\overline{x}_1$, x_2, and x_3. The resulting clause is the empty clause.

(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)

Table 4.3: Example for a resolution proof trace with three literals and four clauses.

A resolution proof is the end product of the CEC process which starts with two circuits, goes on to compute the miter function and then proves the unsatisfiablity of the according Boolean function. In specific cases, the process can also involve transformations of the original problem, i.e. the circuits. Chatterjee et al. demonstrate in [20] how the removal of redundant logic, i.e. rewriting of the logic, and structural hashing techniques do not prevent but actually support the calculation of a single resolution proof output.

I make use of resolution proof traces to set up a proof-carrying hardware scenario for runtime (online) CEC. Figure 4.4 shows the scenario and details the steps that producer and consumer perform for runtime CEC. The consumer decides to order a new reconfigurable hardware module and sends the according specification to a producer. Classically, to create the hardware module the producer utilizes logic synthesis tools which include FPGA technology mapping, and FPGA back end synthesis tools which include place and route and bitstream generation. Additionally, the producer forms a miter from the specification and the hardware module. A CEC tool generates the resolution proof trace. Finally, the producer combines the bitstream and the proof trace into the proof-carrying bitstream and sends it to the consumer. The consumer takes the received proof-carrying bitstream and separates hardware module and proof again. After forming the miter with the hardware module and the original specification, the consumer checks the proof using the proof trace. Only in case the proof holds, the hardware module is loaded into a reconfigurable area of the target device. The work flow is displayed in Figure 4.4. The safety policy consists of the requirement for combinational equivalence of design specification and implementation of the circuit. The validation of the proof is split in two parts as not only the proof needs to be checked for formal correctness but it also needs to match the

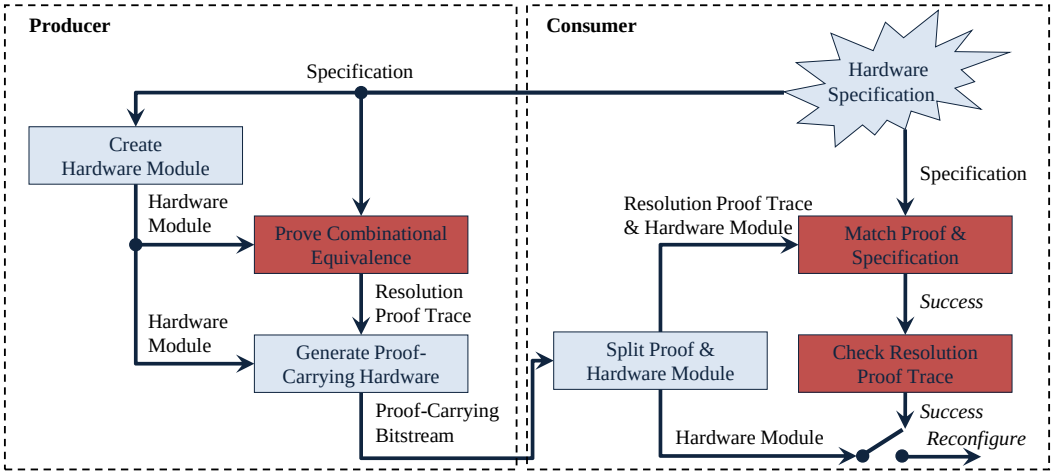


Figure 4.4: Scenario of proof-carrying hardware principle applied to combinational equivalence checks with resolution proof traces as formal safety proof.

hardware module as well as the design specification. For this purpose, the miter function is recomputed with the design specifications and the design extracted from the delivered hardware module and then compared to the miter used for the resolution proof. This step guarantees tamper-proving of the bitstream, as I shall further elaborate in later chapters.

Combinational equivalence between design specification and delivered circuit, i.e. HDL description and synthesized hardware module, is a desirable security assurance for reconfigurable hardware. In this section, I detailed how proof-carrying hardware can utilize resolution proof traces to deliver this safety feature based upon the miter of the respective specification and implementation of a circuit. The computationally costly task of calculating the proof of unsatisfiablity of the miter function, i.e. the resolution proof trace, falls to the producer of the hardware module whereas the consumer only has to check the proof, a lightweight task in comparison. The NP-complete class problem, see [49], of proving unsatisfiability falls to the producer.

4.2 Bounded Sequential Equivalence Checks

The previous section described the merits of proof-carrying hardware applied to combinational equivalence checks to deliver security that lets a consumer establish on-the-fly trust in a newly delivered hardware module from an untrusted source. This section introduces sequential circuits and sequential equivalence of two circuits as security parameter to be verified with the proof-carrying hardware concept. The trust and threat model as well as the merits for the security of reconfigurable hardware devices are comparable to those of Section 4.1.

Clock C	Data D	Q	$\overline{Q}$
rising edge	0	0	1
rising edge	1	1	0

Table 4.4: Truth table for D-type flip-flop. Independent from the previous output value Q_{old}, the new signal D is propagated and stored upon a rising edge of the clock signal C.

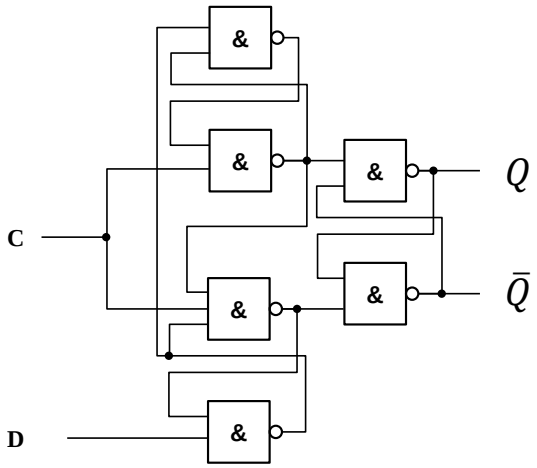


Figure 4.5: The delay flip-flop is constructed using only NAND gates, the clock signal and the data signal to implement the functionality of Table 4.4.

4.2.1 Sequential Circuits

Sequential circuits are an extension of combinational circuits as they rely not only on the current input but also the previous input stored in memory to compute the output. For the purpose of this thesis, the focus is set on synchronous sequential logic, logic which uses a clock signal upon which changes in the storage elements are initiated. For this, the clock rate must be low enough to allows each operation to be completed within one clock cycle, hence the maximum clock rate is determined by the slowest data path in the logic.

Sequential circuits are capable of using back coupling to create memory storage. A basic memory storage unit is a flip-flop (FF), a cross-coupled synchronous circuit with two stables states as memory to store one bit and its negation, the output Q and its negated value $\overline{Q}$. A delay flip-flop, or data flip-flop, has two inputs: the clock signal and the data signal. The new value of Signal Q, see Table 4.4, is changed upon a rising edge of the clock signal and is the same as the value of the data signal D. A circuit can implement the functionality of the truth table by using the previously introduced gates and cross-coupling the signals, as demonstrated in Figure 4.5.

With the ability to store data, a sequential circuit can implement finite state machines.

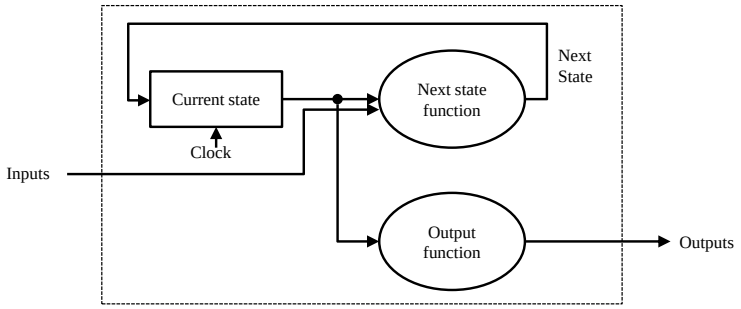


Figure 4.6: A finite state machine as Moore machine, picture from [33].

Finite state machines are a combination of sequential and combinational logic with an input and output function. Mealy machines use the current state and the input to compute the output. In case of a Moore machine, only the new state determines the output of the machine. Moore and Mealy machines are equivalent as one can be transferred into the other one.

A Moore machine, i.e. automaton, $A = \{X, Y, S, \delta, \mu, s_1\}$ is defined by a input alphabet X, an output alphabet Y, a set of states S, a transition function $\delta : S \times X \rightarrow S$ that computes the next state based on the current state and the input, an output function $\mu : S \rightarrow Y$, and a starting state s_1. In Figure 4.6, a generalized Moore machine is depicted. The next state function and output function can be realized with combinational logic, the current state is stored with sequential logic.

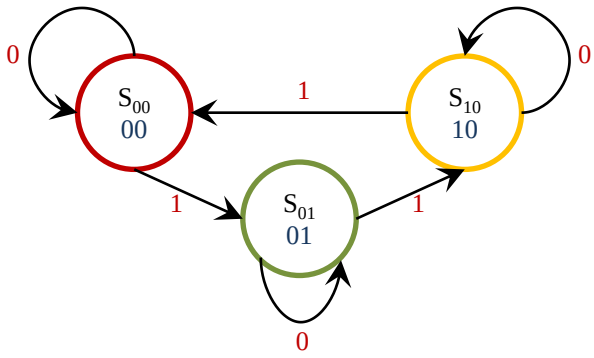
Figure 4.7 is an example of a Moore machine. The purpose of the machine is to light either a red, green, or yellow lamp. The machine operates with three states, the output of each state is the color code. When the input "1" indicates a color switch, the state machine transitions to the next state and thereby changes the color output in the order red - green - yellow -red. Otherwise, with the input signal "0", the current state remains, as does the output. Figure 4.7 shows the Moore machine as a graph and as a table.

4.2.2 Bounded Sequential Miter

Similar to combinational circuits, sequential circuits as well can be equivalent in their design. Two sequential circuits implementing the same state machine offer sequential equivalence without regard to the use of gates, gate inputs, or gate outputs. It is sufficient to prove the equivalence of two state machines to prove the sequential equivalence of the respective circuits. Two state machines A and A' are equivalent if their output is equal in all reachable states:

$$\begin{aligned} &A \equiv A' \text{ if} \\ &\forall s \in S: \quad \exists s' \in S' : s \equiv s' \quad \text{and} \\ &\forall s' \in S': \quad \exists s \in S : s' \equiv s. \end{aligned}$$

The computational effort for this kind of proof grows rapidly with the number of states



state	output	input	next state
S_{00}	00 (red)	0	S_{00}
S_{00}	00 (red)	1	S_{01}
S_{01}	01 (green)	0	S_{01}
S_{01}	01 (green)	1	S_{10}
S_{10}	10 (yellow)	0	S_{10}
S_{10}	10 (yellow)	1	S_{00}

Figure 4.7: This example of a Moore machine uses three states to output the code for either red, green, or yellow light. Blue numbers within the state circle indicate the state's binary output, red numbers on the state transition arrows indicate the state's input.

of the state machines. It is therefore desirable to reduce the problem to a more manageable one. Instead of comparing the complete state machines, it is possible to compare their behavior for a certain limited amount of time frames with bounded sequential equivalence checks. This is a compromise since bounded sequential equivalence checks do not hold the same significance as sequential equivalence checks. Bounded sequential equivalence offers a trade-off between the computational effort and the significance of the proof which can be accepted if the chosen number of time frames is realistic.

With the ABC tool [4] for instance, the miter function of two state machines is build for a fixed number of time frames as described in [59]: For a sequential circuit A, A^1 denotes the combinational part. For n time frames, n copies of A^1 are connected to the FF inputs and outputs. The outputs of this combined circuit are n sets of outputs, one for each of the n time frames. The same is done for circuit B, which has to have the same number of inputs (PI) and FFs. The respective outputs are the input for a logic OR, these are then the input for a single XOR. This means that the behavior of the machines for a fixed number of clock cycles is transformed into a combinational miter as shown in Figure 4.8. An equivalence check is performed based on the unsatisfiability of the combinational miter function as done for combinational equivalence checks in Section 4.1.3.

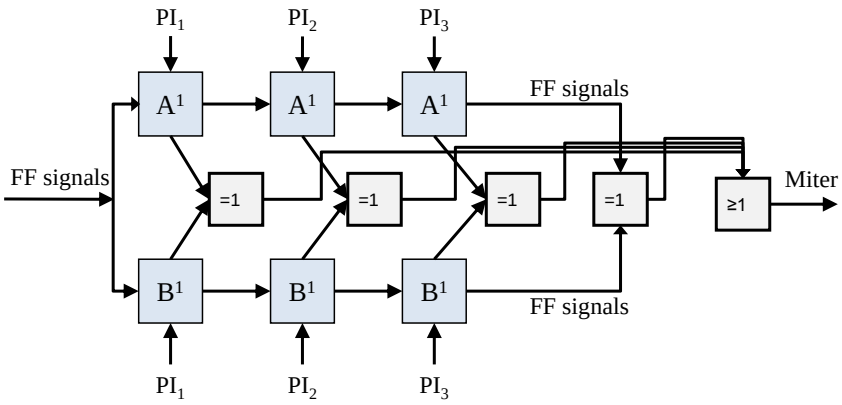


Figure 4.8: Combinational miter based on sequential functions, see [59]. The first set of combinational circuit copies receives the FF signals and the first set of input signals. The content of the FFs is passed along from one combinational copy to its successor.

Listing 4.1: Excerpt from resolution proof in DIMACS CNF format [3] for bounded sequential miter for Verilog source code Listing 4.2.

```
p cnf 94002 245001 // cnf has 94002 variables and 245001 clauses
2 -3 0             // numbers refer to literals
-2 3 0             // negative numbers refer to negated literals
3 -4 31 -33 49 0
3 4 -31 33 -49 0
3 4 31 33 49 0
3 -4 -31 -33 -49 0
-3 -4 33 0         // not(x_3) or not(x_4) or x_33
-3 4 -33 0
-3 31 -49 0
-3 -31 49 0        // not(x_3) or not(x_31) or x_49
4 51042 -5 0
4 51042 6 0
-4 -51042 0
[...]
```

Section 4.2 introduced bounded sequential equivalence checks for sequential circuits, and thereby memory elements and state machines, as an extension to combinational equivalence checks as presented in the previous section. The same benefits of Section 4.1 of shifting the workload to the producer of the module apply as bounded sequential equivalence checks can be performed with resolution proof traces that are quickly to validate by the hardware module consumer.

4.3 Temporal Isolation with Reference Monitors

In Section 2.2.2 and Section 3.4, I mentioned the concept of reference monitors by Huffmire et al. [37] as a novel approach to memory safety through the installation of reference monitor modules on the FPGA chip. This section explains how I employ proof-carrying hardware and combinational as well as bounded sequential equivalence checks as elaborated in Section 4.1 and Section 4.2 to verify and validate the equivalence of reference monitor circuits to their design specifications.

4.3.1 Concept of Temporal Isolation

The threat and trust model, on which the approach of reference monitors and temporal isolation is based, can be summarized in the following scenario, shown in Figure 4.9: On a single FPGA chip, multiple IP cores are installed, all of which need access to an attached off-chip memory. Cores have restricted access to different parts of the memory and access rights that can change dynamically due to memory operations executed by either core. The IP cores cannot be trusted to only access the admissible memory regions or respect access collisions with other cores, hence their access attempts need to be controlled as an unapproved access could impair the memory safety. To govern all access to the external memory, a reference monitor in form of an additional hardware module is installed on the chip to enforce temporal isolation. Temporal isolation in the context of memory safety denotes the isolation of overlapping or unapproved memory accesses due to incorrectly scheduled memory accesses as well as attempts to access memory regions that are at that time off limits to the accessing module. This applies proof-carrying hardware to the use case of runtime monitoring of the target platform. This is an extension of the proof-carrying hardware concept from an application to mere verification and validation to scenarios that require partial verification, for example of only one IP core, as well as monitoring.

4.3.2 Memory Access Policies and Reference Monitors

Reference monitors are compiled from memory access policies defined in a formal language, see [39]. Any memory access policy builds on the basic elements module and range: Modules, i.e. the IP cores on the FPGA chip, request access to memory segments called ranges. Ranges cover the accessible memory and divide it into unique segments of varying length that do not overlap. Each policy is defined by the accesses it allows, each access is defined by its module, range and type of access. The example shows a policy with two different admissible accesses, two different modules accessing each a different range, which can be executed repeatedly.

$$\begin{aligned} \text{Access} &\longrightarrow \text{Module}_1, \text{rw}, \text{Range}_1 \mid \text{Module}_2, \text{rw}, \text{Range}_2; \\ \text{Policy} &\longrightarrow (\text{Access})^*; \end{aligned}$$

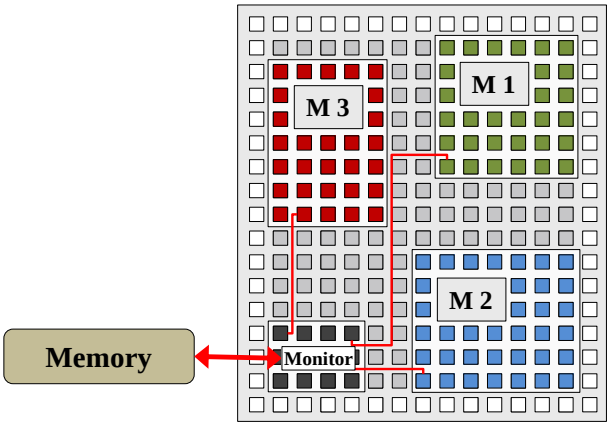


Figure 4.9: Reference monitor module managing memory access requests for external memory.

Modules and ranges as formal concepts allow to create more complex building blocks for policies to express security concepts on a higher level, i.e. in a higher level language. For instance, compartments can contain ranges as well as modules. A module has read access to all memory ranges within its compartment, hence compartments function as equivalence classes assigning the same access rights to all modules within a class. Even more advanced are conflict-of-interest (COI) classes. A COI class contains all ranges that are in conflict with each other. A module can choose one range out of a COI class for access. After that first access, all other ranges within that COI class are off limits for any further access through that module to avoid any conflicts. A module can do this for every COI class as ranges from different COI classes do not create a conflict. Yet another concept for memory access policies described by Huffmire at al., see [37], is the assignment of different security levels to ranges and modules. Four security classifications describe the range of trust from Unclassified (U) to Classified (C) and Secret (S) up to Top Secret (TS), a standard classification also used by the government of the United States of America, see [6, 7]. Memory access requests can be denied or granted on the base of the security classification of the module and memory range, respectively. The use of these concepts within a memory access policy described in a higher level language is demonstrated in Table 4.5.

With those and other concepts, many different memory access policies can be formed to implement different memory security scenarios. Policies can also be dynamic, which allows them to change access rights at runtime. I chose six different memory access policies, three static and three dynamic ones as elaborated in [37], to investigate the possible application of the proof-carrying hardware concept:

These are the three static types of memory access policies used for this thesis:

- The Isolation (Iso) model assigns every memory range and every module to compartments. Within a compartment, modules have read-only access to the ranges in

the same compartment.

- The Bell and LaPadula (BL) model is a confidentiality model which prevents modules from reading memory areas with higher security classifications (no read-up) and from writing into memory ranges with lower classifications (no write-down).

- The Biba (Biba) model is made to protect the integrity of the data. It prevents modules from writing into memory ranges with a higher security classification (no write-up) and from reading data from a range with a lower security classification (no read-down).

These are the three dynamic types of memory access policies used for this thesis:

- The Chinese Wall (Chin) model is build upon conflict-of-interest classes. Each range is a member of a conflict-of-interest class. A module is granted read access to one range of its choosing of each class.

- The High Watermark (High) model realizes data confidentiality similar to the Bell and LaPadula model but also extends that concept by allowing write-downs. After a write-down, the security classification of the range is changed to the higher security level of the module that performed the access. This excludes all modules with a lower security classification from further accessing that memory range.

- The Low Watermark (Low) model realizes data integrity similar to the Biba model but extends it by permitting read-downs. Upon a read-down, the module is assigned the lower classification of the range that it just accessed, thus changing the access rights for that module.

Table 4.5 shows instances of four policies. The first two, a) the Isolation (Iso) and b) the Bell and LaPadula (BL) model, are both static. The next two, c) the Chinese Wall (Chin) and b) the High Watermark (High) are dynamic.

The Isolation (Iso) model features two compartments that manage a total of three ranges and three modules. Module_1 has read-only access to $\text{Range}_{1,2}$, $\text{Modules}_{2,3}$ are granted read-only access to Range_2.

The Bell and LaPadula (BL) model is executed with two modules and four ranges where each range and each module is assigned a security classification. The objective of this model is to protect the data confidentiality. Module_1 is registered as "Unclassified"(U), the lowest security classification. Hence, it cannot read $\text{Ranges}_{2,3,4}$ but can write into each range. Module_4 is designated as "Top secret"(TS), the highest classification. Therefore, it cannot write into $\text{Ranges}_{1,2,3}$ as those ranges could be read by modules with a lower classification. But opposed to Module_1, Module_4 can read all ranges.

The Chinese Wall (Chin) model is focused on avoiding conflicts between memory ranges and assigns each range to a COI class. The instance given below has three COI classes containing two ranges each. The subject, i.e. module, can choose one range from each class to access. That means a maximum of three ranges to access, but all of them have

a)

Iso;
$\text{Compartment}_1 \longrightarrow \text{Module}_1$;
$\text{Compartment}_1 \longrightarrow \text{Range}_1$;
$\text{Compartment}_1 \longrightarrow \text{Range}_2$;
$\text{Compartment}_2 \longrightarrow \text{Module}_2$;
$\text{Compartment}_2 \longrightarrow \text{Module}_3$;
$\text{Compartment}_2 \longrightarrow \text{Range}_3$;

b)

BL;
$\text{Module}_1 \longrightarrow$ U;
$\text{Module}_2 \longrightarrow$ TS;
$\text{Range}_1 \longrightarrow$ U;
$\text{Range}_2 \longrightarrow$ C;
$\text{Range}_3 \longrightarrow$ S;
$\text{Range}_4 \longrightarrow$ TS;

c)

Chin;
$\text{COI\_Class}_1 \longrightarrow \text{Range}_1$;
$\text{COI\_Class}_1 \longrightarrow \text{Range}_2$;
$\text{COI\_Class}_2 \longrightarrow \text{Range}_3$;
$\text{COI\_Class}_2 \longrightarrow \text{Range}_4$;
$\text{COI\_Class}_3 \longrightarrow \text{Range}_5$;
$\text{COI\_Class}_3 \longrightarrow \text{Range}_6$;
$\text{Subject} \longrightarrow \text{Module}_1$;

d)

High;
$\text{Module}_1 \longrightarrow$ U;
$\text{Module}_2 \longrightarrow$ TS;
$\text{Range}_1 \longrightarrow$ U;
$\text{Range}_2 \longrightarrow$ C;
$\text{Range}_3 \longrightarrow$ S;
$\text{Range}_4 \longrightarrow$ TS;

Table 4.5: Examples of static memory access policies Isolation (Iso) and Bell and LaPadula (BL) and dynamic policies Chinese Wall (Chin) and High Watermark (High).

to be in different classes. Admissible combinations of memory ranges include, but are not limited to: $\text{Range}_{1,3,5}$, $\text{Range}_{2,4,6}$, $\text{Range}_{4,5}$, Range_2.

The High Watermark (High) model is the dynamic version of the Bell and LaPadula model. Hence, it is described exactly the same way except for the policy title. The resulting policy reacts to dynamically to write-downs, though. Module_2 change the security classifications of $\text{Range}_{1,2,3}$ to TS by writing into them, for example.

The reference monitors compiled from the policies are state machines. The static policies result in state machines with a single state (not counting an error state), which are effectively combinational circuits. The dynamic policies result in state machines with multiple states, i.e. sequential circuits. Those static and dynamic hardware modules are the test functions for the tool flow, elaborated in Chapter 5.

The actual reference monitor hardware modules are described in the Verilog HDL. In [37], Huffmire et al. elaborate the process of synthesizing a Verilog hardware module from a policy. The hardware synthesis is a process combined of several steps. In Table 4.6 a), an instance of the Chinese Wall (Chin) policy is depicted. The example uses one module that can access one of the two memory ranges of the one COI class repeatedly. Given the policy in Table 4.6 a), the compiler performs the following actions:

- It constructs a syntax tree from the policy, converts it into an expanded intermediate form and then translates the policy to a regular expression, see Table 4.6 b).

a)

Chin;
$\text{COI\_Class}_1 \longrightarrow \text{Range}_1$;
$\text{COI\_Class}_1 \longrightarrow \text{Range}_2$;
Subject $\longrightarrow \text{Module}_1$;

b)

((Module1ReadsRange1*)|(Module1ReadsRange2*))

c)

$\text{Access}_0 \longrightarrow (\text{Module}_1\text{ReadsRange}_1)^*$;
$\text{Access}_1 \longrightarrow (\text{Module}_1\text{ReadsRange}_2)^*$;
Policy $\longrightarrow \text{Access}_0 \mid \text{Access}_1$;

Table 4.6: Instance of Chinese Wall policy. Part a) depicts the high level language description of the policy, b) presents the resulting regular expression. Part c) gives the translation of the policy in the lower level language used for the last steps of the policy to Verilog compilation.

The regular expression describes the same scenario as before: Module_1 is granted indefinite read access to either Range_1 or Range_2.

- It transforms the regular expression to a non-deterministic finite automation (NFA) and converts the NFA to a minimized deterministic finite automaton (DFA), Figure 4.10 and Figure 4.11.
- It converts each range into a covering set of aligned power of two ranges.
- It outputs the range detection and translation of the policy into the lower level language as shown in Table 4.6 c), then delivers the state machine logic as synthesizable Verilog, see Listing 4.2.

The Verilog source code for the Chinese Wall reference monitor module is depicted in Listing 4.2. The source code structure is similar for all policies: The input contains, besides the clock and a reset signal, information regarding which module requests what type of access to which memory range. The output is a single bit, indicating whether a request is approved or denied. The reference monitor's work is performed by two processes, one for realizing the output function and one for implementing the state machine. The state machine process is initiated with a "always @(posedge clock or posedge reset)" and therefore monitors the clock and reset signal. It reacts when one of them changes from low to high as indicated by the word "posedge" which denotes the positive edge of the signal. As stated earlier, there is a difference between dynamic and static policies. Static policies are not capable of changing any access rights at runtime. Hence, they only have one regular state where requests are processed. The process governing the output function is initiated with a "always @(state)" and reacts upon any state changes. As long as the state machine's state is considered valid, the memory access is granted.

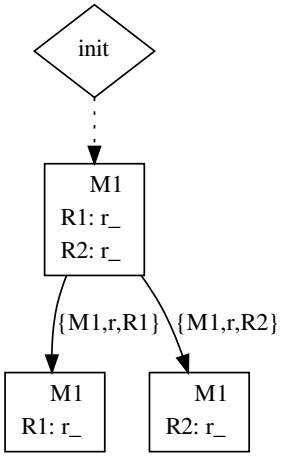


Figure 4.10: State machine-like representation for policy specified in Table 4.6 a). After initialization, Module_1 has read access to both modules but no write access as indicated by "r_". After the first read access performed by Module_1, all further accesses are limited to reading either Range_1 or Range_2.

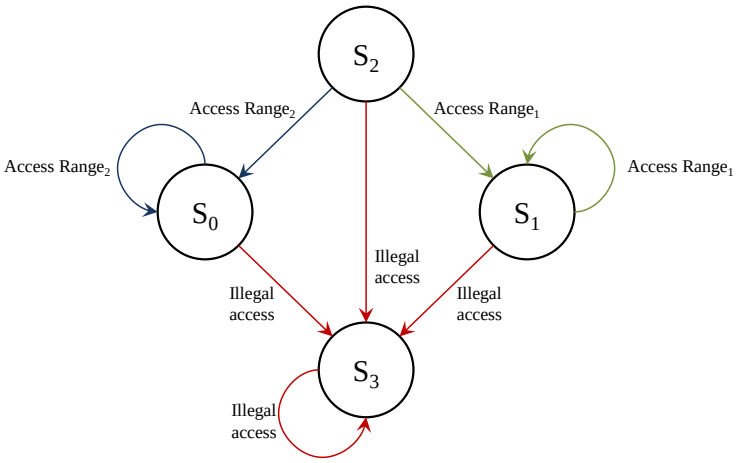


Figure 4.11: DFA for state machine Listing 4.2, including the error state S_3 and omitting the non-state rhombus indicating the initialization.

As shown in Figure 4.10, the state machine described by Listing 4.2 has three regular states. State s2 is the first state after the initialization. Module_1 has not yet issued any memory access request and can still choose to read-only from either Range_1 or Range_0. The two valid read requests are highlighted with red comments. Any one of those two possible read accesses will change the state from s2 to either s1 or s0 where the according read access can be repeated. The state machine will switch to state s3 and remain there and not process any request anymore if at any point an invalid memory access request is

being made. This state is not pictured in Figure 4.10 as it is merely a construct of the Verilog code but not an actual state of the state machine resembling the policy. A regular depiction of the state machine is Figure 4.11.

Reference monitor hardware modules are an efficient way of enforcing memory access policies on an FPGA chip where multiple IP cores require access to shared memory. This approach to memory safety is versatile due to the different access scenarios available as memory access policies. The formal high level language to specify memory access policy makes use of the concepts of modules as subjects requesting access to memory ranges and creates more advanced formal concepts based on modules and ranges to allow easy specification of memory access policies. In this section, I reviewed various dynamic and static memory access policies, the according compiler, and explained the different underlying principles of the according safety scenarios as developed by Huffmire et al. in [40, 37].

Listing 4.2: Verilog source code for Chinese Wall (Chin) reference monitor specified in Table 4.6.

```
module State_Machine(clock,reset,module_id,op,address,is_legal);
// global inputs and output; local registers, parameters, wires
  input clock, reset;
  input [4:0] module_id;
  input [1:0] op;
  input [31:0] address;
  output is_legal;
  reg is_legal;
  reg[1:0] state;
  parameter s0 = 'd0;
  parameter s1 = 'd1;
  parameter s2 = 'd2;
  parameter s3 = 'd3;
  wire r0;
  wire r1;
  assign r0=(address[31:4]==28'd1)?1'b1:1'b0;
  assign r1=(address[31:4]==28'd2)?1'b1:1'b0;

  always @(state) // check output signal after every access
  begin
    case (state)
      s0:
        is_legal=1'b1; // grant access for states s0, s1, and s2
      s1:
        is_legal=1'b1;
      s2:
        is_legal=1'b1;
      s3:
```

```
        is_legal = 1'b0; // deny access for error state
      default:
        is_legal = 1'b0; // deny access by default
    endcase
  end

  always @(posedge clock or posedge reset) // check access
                    // rights at every clock cycle or reset
  if (reset) state = s2;
  else
    case (state)
      s0:  // Module_1 accesses Range_2
        case({module_id,op,r0,r1}) // module, access type, range
          9'b000010101:
            state = s0; // access Range_2
          default:
            state = s3; // error state
        endcase
      s1: // Module_1 accesses Range_1
        case({module_id,op,r0,r1})
          9'b000010110:
            state = s1; // access Range_1
          default:
            state = s3; // error state
        endcase
      s2: // state after initialization, access to either region
        case({module_id,op,r0,r1})
          9'b000010101:
            state = s0; // access Range_2
          9'b000010110:
            state = s1; // access Range_1
          default:
            state = s3; // error state
        endcase
      s3: // error state - no state transition, only reset
        state = s3;
      default:
        state = s3;
    endcase
endmodule
```

4.4 Chapter Conclusion

In this chapter, I elaborated three major security concepts for reconfigurable hardware. Equivalence between design specification and synthesized hardware is an important security property for (dynamically) reconfigurable hardware as it attests that the functionality of the hardware is the same as its design. For combinational circuits, a combinational equivalence check builds a miter function out of the two circuits in question. The unsatisfiability, which guarantees the equivalence, is proven with a resolution proof trace. True to the concept of proof-carrying hardware, that costs to compute the proof can be high but comparatively low to validate the proof. This allows the consumer of hardware modules from untrusted source to establish trust in the newly received IP core.

Sequential circuits are an extension to combinational circuits as they offer to ability to store data and hence are capable of implementing state machines and memory. The equivalence between two sequential circuits is the equivalence of the output of their respective state machines. To render the task of proving sequential equivalence more manageable, I follow the concept of bounded sequential equivalence checks. They compare the behavior of sequential circuits over fixed and sufficiently large number of clock cycles to construct a miter function. As done before for combinational equivalence checks, the miter function is proven unsatisfiable with resolution proof traces.

Reference monitors secure shared memory by managing all access request issued by multiple IP cores on a single FPGA chip. Memory access policies describe the access rights of the individual hardware modules on the chip with regard to the individual ranges of the shared memory. A compiler then produces the Verilog source code for further synthesis of the hardware. A major benefit of this approach by Huffmire et al [40, 37] to spatial isolation of hardware modules is the flexibility to simply exchange the single reference monitor IP core to enforce a novel policy and leaving the remaining IP cores as they were.

These three approaches to increase security for reconfigurable hardware and the reconfiguration of such are particularly suitable for the proof-carrying hardware concept. Equivalence checks allow for a validation of the proof by the consumer despite possible platform restrictions, hence enabling fast or even on-the-fly trust. The application of proof-carrying hardware to the verification of reference monitors is the application of proof-carrying hardware to scenarios that require monitoring of security properties of the target platform during runtime. This demonstrates that proof-carrying hardware is a flexible concept which can establish security in different security critical scenarios.

The prototype tool flow elaborated in Chapter 5 implements the application of proof-carrying hardware to these verification challenges.

CHAPTER 5

Evaluation Methodology

This chapter details the evaluation methodology for my novel proof-carrying hardware approach. My methodical approach is the use and adaption of open-source tools for the development of a proof-carrying hardware prototype tool flow. This prototype tool flow will serve to evaluate the thesis claim made in Section 3.5 as it is the implementation of proof-carrying hardware applied to equivalence checks of combinational and sequential reference monitors. The advantage of open-source tools over commercial tools is the potential of open formats. The full access to input and output formats enables me to define the format of a proof-carrying bitstream and according FPGA architecture, see [29]. This advantage outweighs the advantage of using proprietary tools and formats which would allow for an application of proof-carrying hardware to real-life reconfigurable devices, such as the Xilinx Virtex-4 [41].

In Section 5.1, I introduce an abstract FPGA architecture and matching bitstream format. The Bitstream Composer / Decomposer tools form together with open-source tools a tool flow implementing a proof-carrying hardware prototype. The tool flow given in Section 5.2 is the prototype implementation of the proof-carrying hardware. It realizes the proof-carrying hardware principle with a split in the work flow between consumer and producer of hardware modules.

5.1 Architecture and Bitstream Format for Abstract FPGA

I now present my abstract FPGA architecture and according bitstream format for the tool flow that will be elaborated in this chapter.

The FPGA is composed of global inputs, global outputs, logic blocks including Look-Up Tables (LUT) or latches, and connections between those components, displayed in Figure 5.1. The size and shape of the rectangular array of logic blocks is variable. The number of logic blocks in y-direction of the array also gives the number of global inputs and outputs of the FPGA chip, as each block in the first column of the logic block ar-

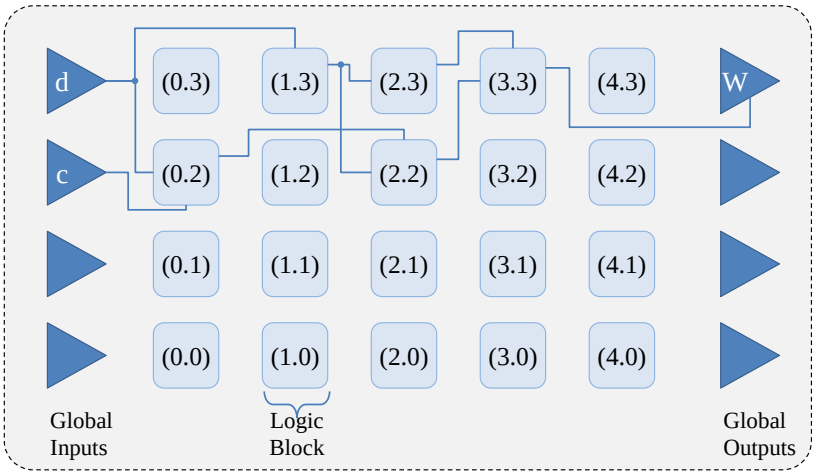


Figure 5.1: Abstract FPGA architecture.

ray is connected to a global input and each block in the last column is connected to a global output. Global inputs and outputs are individually assigned or intentionally left unassigned.

A logic block consists of either a Look-Up Table (LUT) and pins or a latch and pins that are connected to incoming or outgoing signals. Any logic block can have either a single global output, that is a signal routed to a global output pin, or a local output, which is a signal routed from one logic block to one or more other logic blocks. The name of the logic block is at the same time its position in the array of logic blocks. The positions are assigned from left to right and from bottom to top, which is a coordinate system scheme similar to the one used by VPR, see [11]. All LUTs have the same number of inputs, which is variable and determined by the largest number found among all logic blocks. For the listed input signals, a truth table defines the programming of the LUT. A latch has only one input signal which it stores.

The FPGA architecture is basic as it is meant for later extension as the need arises. One aspect of later extensions are the logic blocks. A type of logic block which combines the capabilities of the LUT type logic block and the latch is plausible. Also, functional blocks such as memory units and dedicated multipliers can be part of an extended architecture as they are part of modern FPGA chips. Such a design decision will have to depend on the context of synthesis tools and hardware device for which the architecture is modified. Another possible focus for an extension of the abstract FPGA architecture is the routing. The current routing is capable of specifying which component, i.e. global I/O or logic block, connects to which other component. Other than the designated pin, the routing does not assign routing resources as those are not specified in the design. A specific description of the routing resources and their usage could be part of the design.

The abstract FPGA architecture goes together with the according bitstream format. The bitstream format for my FPGA architecture captures the above specifications implic-

itly and explicitly, see Listings 5.1 for an exemplary excerpt. The preamble states general information regarding its design. The dimension of the logic block array and the number of inputs for each LUT are explicitly stated. A list names and assigns all global inputs on the chip from bottom to top, a "–" marks the global inputs on the chip that are not assigned and therefore remain unused. A second list specifies the names and locations of the global outputs the same way. Hence, the inputs and outputs are named explicitly, their order is defined implicitly by the order in which they are listed. A description of each LUT or latch logic block makes up the second part of the bitstream. Each block, LUT as well as latch, is defined by its input and output. As the LUT may have more inputs available than what is actually needed, the format marks unused inputs with an "–" and utilized designated input signals in their specified order. A latch has only the one input signal by design. The connections for the block's output are listed separately at the end of each block. In case of a LUT logic block, the block describes the programming of the LUT with a truth table. If the LUTs to be programmed are of size n, the table of each logic block has 2^n entries. This second part of the bitstream, the list of all logic blocks, defines not only each logic block's functionality. It also reveals the amount of configured LUTs and latches logic blocks and the number of routing connections between the components.

I designed the abstract FPGA architecture and accompanying bitstream format presented here to be incorporated in the proof-carrying hardware tool flow presented in the following section. As pointed out earlier, the architecture and its bitstream format are custom made for the this purpose but they are flexible enough to accommodate later extensions and changes to adopt for different tasks of the tool flow.

Listing 5.1: Excerpt from bitstream file Listing A.5 for Chines Wall Verilog example Listing 4.2.

```
.begin

block_array_size: 8x8
lut_size: 2

gin:  clock reset module_id0 \
module_id1 module_id2 \
module_id3 module_id4 op0 op1 \
adr0 adr1 adr2 adr3 adr4 adr5 \
adr6 adr7 adr8 adr9 adr10 \
adr11 adr12 adr13 adr14 adr15 \
adr16 adr17 adr18 adr19 adr20 \
adr21 adr22 adr23 adr24 adr25 \
adr26 adr27 adr28 adr29 adr30 \
adr31
// all global inputs

gout: is_legal
// all global outputs

.gout is_legal (7,0).6
// global output routing

.gin clock (3,9)
lout (8,3).default
lout (8,5).default
// special clock routing

.gin reset (9,4)
lout (8,4).3
lout (8,5).3
// regular global input routing

.gin module_id0 (0,3)
lout (1,3).2

.gin module_id1 (0,3)
```

```
lout (1,3).1

.gin module_id2 (6,0)
lout (6,1).3

.gin module_id3 (6,0)
lout (6,1).0

.gin module_id4 (5,0)
lout (5,1).3

.gin op0 (0,2)
lout (1,2).3

.gin op1 (0,2)
lout (1,2).2

.gin adr4 (0,5)
lout (1,5).1
lout (1,4).0

.gin adr5 (0,5)
lout (1,5).2
lout (1,4).2

[...]

.latch (8,5)
n151 FF_NODE33
lout (7,3).0
lout (6,2).3
lout (6,3).2
// latch logic block
// names of input and output
// multiple local outputs

.latch (8,3)
n191 FF_NODE34
lout (7,3).2
lout (6,2).2

.lut (7,3)
FF_NODE33 FF_NODE34 n48
11 1
00 1
lout (5,2).1
lout (7,1).3

.lut (1,1)
adr26 adr27 n49
00 1
lout (2,1).1
// lut logic block
// names of inputs and outputs

.lut (4,1)
adr25 adr31 n50
00 1
lout (2,1).2

[...]

.lut (7,5)
n75 n88 n89
11 1
lout (7,4).2
lout (8,5).2

.lut (8,5)
reset n89 n151
00 1

.lut (6,2)
FF_NODE33 FF_NODE34 n91
lout (5,2).0

.lut (5,2)
n48 n91 n92
00 1
lout (5,3).3

.lut (5,3)
n81 n92 n93
11 1
lout (5,4).3
```

```
.lut (5,4)
n75 n93 n94
01 1
lout (6,4).2

.lut (6,4)
n86 n94 n95
11 1
lout (7,4).1

.lut (7,4)
n89 n95 n96
00 1
lout (8,4).2

.lut (8,4)
reset n96 n97
00 1
lout (8,3).2

.lut (7,1)
- n48 is_legal
// first input is not used
-0 1
gout (7,0).6
// global outputs

.lut (8,3)
- n97 n191
-0 1

.end
```

5.2 Open-Source Prototype

Firstly, this section elaborates on the work steps of the prototype tool flow and the tools that perform those work steps. Secondly, I detail the prototype and the role split between the consumer and the producer of the proof-carrying bitstream as this split is inherent to the proof-carrying hardware principle.

The proof-carrying hardware prototype implements the scenario pictured in Figure 3.2. The description of the desired hardware module's functionality is a design specification file in the Verilog format; the safety policy consists of a single requirement: equivalence of the specification file and the circuit included in the final bitstream. Due to a lack of any verification for the necessary translation from Verilog to the blif format [2, 16], the blif format serves as design specification. The type of equivalence depends on the type of circuit to be validated, i.e. combinational equivalence as in Section 4.1 or bounded sequential equivalence as in Section 4.2. The circuits to be checked for equivalence are dynamic and static reference monitors as detailed in Section 4.3. In general, any format that consumer and producer agree upon could be used to convey the design specifications instead of Verilog / blif. For this thesis, the use of ODIN II suggests the use of Verilog files and their respective translation into the blif format. The type of proof used to validate the combinational equivalence is a resolution proof, see [28, 54, 20], which is attached to the bitstream and makes it a proof-carrying bitstream. An overview of the tool flow, consisting of a tool flow for the consumer and producer each, is given in Figure 5.2:

5.2.1 Open-Source Tools

I now present the open-source tools which contribute to the prototype testing environment for proof-carrying hardware. The prototype proof-carrying hardware tool flow, as detailed in Figure 5.2 employs open-source tools for the hardware module production as well as the verification and validation process:

- Odin II [58] by Jamieson et al. (University of Miami) performs file format conversion and front-end synthesis from circuit specification in Verilog to a logic function in blif (Berkeley Logic Interchange Format).

- ABC [19] by Mishchenko et al. (Berkeley) is a "A System for Sequential Synthesis and Verification".
  - It performs logic optimization and technology mapping to Look-Up Tables (LUTs), generating again a logic function (blif file). The optimization is based on And-Inverter Graphs (circuits composed exclusively of two-input AND gates and inverters to negate signals) and includes balancing, refactoring and rewriting; all applied multiple times to the circuit with the resync2 command. Balancing performs a minimum delay tree-decomposition of each AND gate to reduce the delay of the And-Invertor Graph (AIG) measured in logic levels of two-input AND gates to a minimum. Refactoring collapses and refactors the logic cones in the AIG and thereby reduces the number of AIG nodes as well as the number of logic levels, which is also the aim of the DAG-aware Rewriting as described in [54].
  - It also forms the miter cnf, see Section 4.1.2, with the previously extracted logic function and the design specification by translating the cnf from And-Inverter Graphs. Depending on the type of circuit, one of two different miter types is created: In case of a combinational reference monitor, a simple combinational miter is build as described in Section 4.1.2. If the test function is sequential, then a sequential miter is build and unrolled for 1000 time-frames, see bounded sequential equivalence checks as described in Section 4.2, which results in a combinational miter.

- T-VPack [14, 11] by Betz et al. (University of Toronto) packs the LUTs into logic blocks, generating a circuit description in form of a FPGA netlist (.net) of logic blocks. T-VPack converts the LUT netlist into the format required by VPR.

- VPR [14, 11] by Betz et al. (University of Toronto) places and routes the netlist and produces placement (.p) and routing (.r) information. The target architecture input file used by VPR has to agree with the specifications used for T-VPack. The placement and routing is performed as elaborated in Section 2.1.2.

- The Bitstream Composer creates the bitstream in the format according the FPGA architecture as given in Section 5.1. The three essential types of information that

make up the bitstream content are the logic function of the circuit, the placement information, and routing information. I developed this tool as part of this thesis.

- The Bitstream Decomposer extracts the logic function, the placement information, and the routing information from the bitstream composed by the Bitstream Composer. I developed this tool as the counterpart to the Bitstream Composer.
- The SAT solver PicoSAT [17] by Biere et al. (Johannes Kepler University) proves the unsatisfiability of the miter which is equivalent with the combinational equivalence of design specification and delivered hardware module by generating an extended proof trace.
- TraceCheck [1] by Biere et al. (Johannes Kepler University) validates the correctness of a proof trace. The output is a compact binary resolution trace that demonstrates the resolutions steps necessary to derive that the proof is indeed unsatisfiable.

The first four tools of the list, Odin II, ABC, T-VPack, and VPR, are a common open-source tool chain for hardware synthesis. In combination with the remaining tools to verify and validate equivalence to design specifications and create a bitstream, all tasks in the tool flow have been covered.

5.2.2 Consumer and Producer Tool Flow

One principle of proof-carrying hardware is the usability elaborated in Section 3.3. The usability of proof-carrying hardware means to shift the majority of the workload to the producer, i.e. the untrusted source of the hardware module. Hence, the prototype as depicted in Figure 5.2 consists of tasks assigned to the producer and tasks performed by the consumer. This separation shifts the majority of the workload to the producer by assigning the smallest amount possible of the workload to the consumer.

The tasks in the tool flow Figure 5.2 can be grouped according to Figure 3.2. It is assumed that the consumer has already produced the design specification, i.e. logic function, to describe the safe behavior of the resulting hardware module. The producer has to perform the following tasks:

- Create Hardware Module:
  1. The Verilog hardware description of the circuit is synthesized into a blif logic function.
  2. The blif logic function undergoes logic optimization and technology mapping to LUTs, resulting in a blif file.
  3. The logic function is transformed into a netlist.
  4. The netlist is placed and routed according to the FPGA's architecture.
- Generate Hardware Bitstream:
  The bitstream is compiled and includes the logic function, placement and routing information.

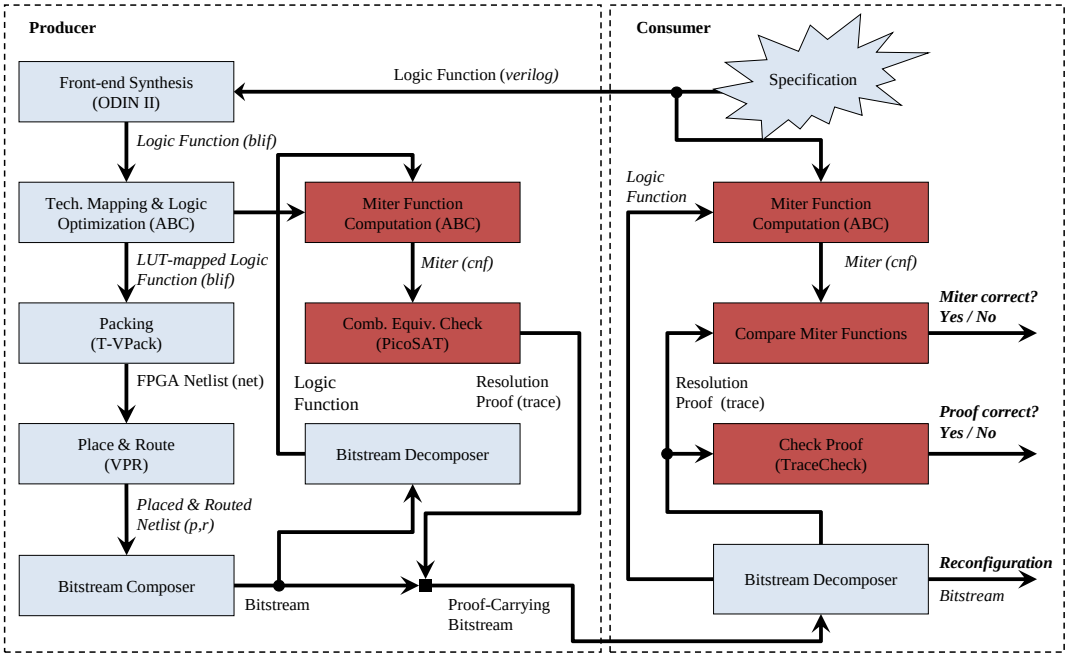


Figure 5.2: Complete tool flow to perform hardware module creation (from synthesis to place & route) as well as the combinational equivalence check and proof validation.

- Proof Safety of Module:
  The bitstream is decomposed and the logic function is extracted. A miter and resolution proof computation for either a combinational or bounded sequential equivalence check takes place.

The proof-carrying bitstream is then sent to the consumer who performs the following tasks:

- Extract Proof:
  The bitstream is separated into the safety proof, the logic function of the hardware module, and the placement and the routing information of the hardware module.

- Check Proof:
  1. The miter is recomputed using the original design specification and the extracted logic function.
  2. The newly computed miter is compared against the miter contained in the delivered proof.
  3. The proof is checked for formal correctness.

These two bullet point lists along with Figure 3.2 clearly state the separation of tasks that fall to the untrusted source of the hardware module and the host of a reconfigurable device. The resulting prototype tool flow is shown in Figure 5.2.

This concludes the setup of the prototype tool flow which is used for evaluating the usability and robustness as principles of proof-carrying hardware, see Section 3.3, which is depicted in the following Chapter 6.

5.3 Chapter Conclusion

This chapter detailed my methodical approach for evaluating the novel proof-carrying hardware concept which is implemented using an open-source prototype tool flow to evaluate the thesis claim made in Section 3.5. As mentioned at the beginning of this chapter, the advantage of open-source tools over commercial tools is the potential of open formats which enabled me to define a bitstream format for the prototype tool flow. Despite these advantages of open-source tools and formats, it is still desirable to transfer proof-carrying hardware additionally to proprietary FPGAs and bitstreams. This would demonstrate how well this novel concept applies to real-life security challenges and scenarios.

The prototype implementation of proof-carrying hardware serves as a proof of concept as it covers the verification of a range of diverse security issues. It offers the split between the untrusted producer and consumer of a new hardware module which enforces the the shift of workload and computational burden to the producer, as I will demonstrate in Chapter 6 where I will also demonstrate the robustness of the prototype.

CHAPTER 6

Experimental Results

This chapter discusses the experimental results performed with the proof-carrying hardware prototype tool flow. The experimental validation of the prototype focuses on the integral aspects of the proof-carrying hardware approach: flexibility, robustness and especially the usability, as explained in Section 3.3. In Section 6.1, the robustness against certain attacks is demonstrated as well as limitations to the security that proof-carrying hardware aims to offer. In Section 6.3.1 and Section 6.3.2, I give conclusive evidence of how the prototype indicates the general usability of proof-carrying hardware by depicting the runtime and memory usage comparison for the consumer and producer parts of the tool flow as introduced in Chapter 5 to demonstrate the desired shift of workload from consumer to producer.

6.1 Robustness and Limitations

6.1.1 Robustness

In the previous section, I elaborated an experimental setup: a tool flow and FPGA architecture for a proof-carrying bitstream that contains a hardware module for a reconfigurable system. I claim that this setup is sufficient to verify the tool flow and transmission. The process executed by the prototype can be grouped into several steps. For each, I consider how proof-carrying hardware approach would detect a failure to comply to the safety policy of combinational equivalence:

1. Scenario 1: Manipulation of Hardware Production
   Proof-carrying hardware has to ensure that any changes to the circuit through the tools themselves, marked with a 1 in Figure 6.1, would be detected. ODIN II is not included, as for the intent and purpose of this work the verification of the Verilog to blif translation is omitted due to a lack of readily available validation tools. If

the logic function is submitted to combinational changes during the first step in block 1, those changes would be carried through all further steps and be conveyed into the bitstream. The following tools convert the logic function into a netlist that is packed, placed, and routed. If there is any other input used or the module altered, the computation of the bitstream will fail as the logic function would not match the placement and routing information. If a bitstream composition was still successful, the extracted logic function would convey the changes and not allow for a successful miter computation or result in a different miter from what the consumer will later calculate. This capability to integrate and extract any information as needed is possible with my own bitstream format for my FPGA architecture introduced in Chapter 5.

2. Scenario 2: Manipulation of the Proof Computation
   The producer uses two tools for the computation of the formal proof. One for creating the miter and one for computing the resolution proof trace, marked with a 2 in Figure 6.1. Since the miter is recomputed by the consumer, any kind of error in that step would be detected by the difference between the miters. The producer is also in no position to use even the hardware module's logic function from any intermediate (correct) step instead of bitstream extraction to achieve deceit. The miter might be unsatisfiable and result in a correct resolution proof, but it would still differ from what the consumer expects as his miter is build upon the bitstream extraction. Therefore, it would also not be possible to take a different design specification in Verilog translated to blif format as input for this step. The same arguments apply to the computation of the resolution proof itself. If the proof is not correct or uses a different miter, the consumer will notice this upon comparing the miter and checking the proof. This eliminates the possibility to use any other than the appropriate miter function or to pass on an incorrect resolution proof.

3. Scenario 3: Manipulation of Proof-Carrying Bitstream
   A third party might maliciously intercept the proof-carrying bitstream between consumer and producer, marked with a 3 in Figure 6.1. If such a man-in-the-middle attack corrupts the resolution proof, the proof simply would not check out anymore and be dismissed by the proof checker in the consumer's tool flow. If, by any chance, the proof might still be a correct proof, it would not match the bitstream anymore. This would also be detected because the consumer recomputes the miter function and matches it to the miter that is included in the resolution proof. Even if the hardware module within the bitstream is changed according to the changes in the resolution proof, the miter would differ or be satisfiable since the original logic function is used by the consumer to rebuild the miter. Similar, if only the hardware module section of the proof-carrying bitstream is manipulated, the resolution proof would not match the miter computed with the extracted logic function.

As with all verification problems, a prerequisite is that the tools to validate the soundness of the formal proof function correctly. This regards the proof checking, miter computation,

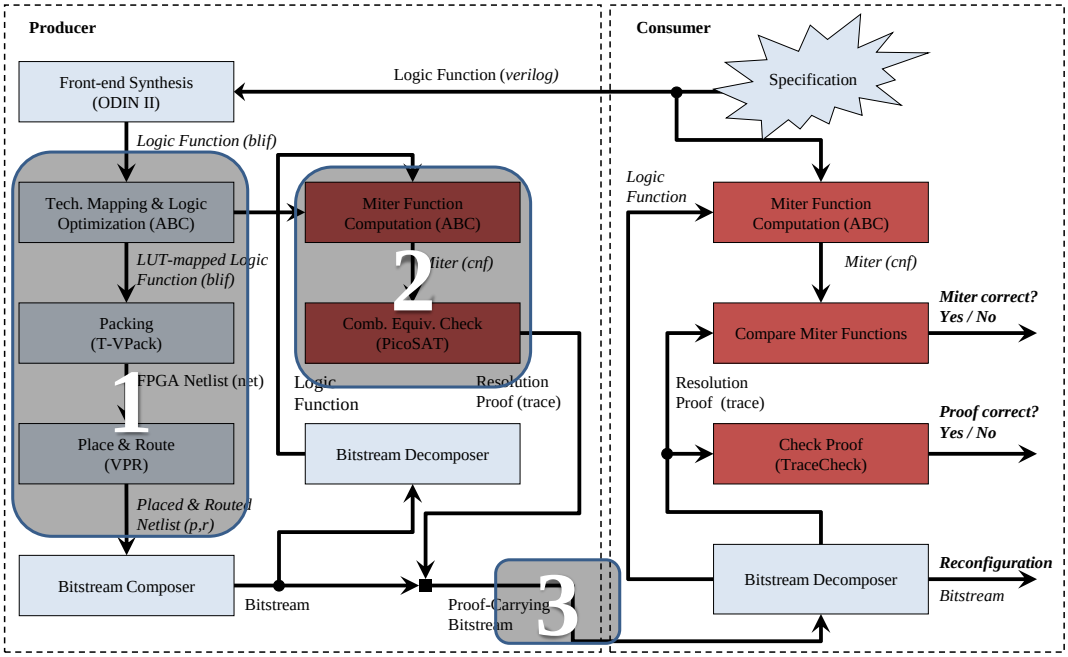


Figure 6.1: Validation of Tool Flow (sketch of tool flow as in Figure 5.2).

and matching of the miter functions performed by the consumer. Also, the extraction of the logic function from the bitstream has to be done correctly. The correct functionality of the tools for these tasks is the trusted base upon which the proof-carrying hardware prototype operates. This way, as discussed in Chapter 3, proof-carrying hardware neither relies on secure transmission nor on any previously established trust in a producer. By establishing trust in the delivered product, security can still be guaranteed without establishing trust in the producer of a hardware module or employing encryption.

6.1.2 Limitations

A natural limitation of proof-carrying hardware is the completeness of the safety policy used to set the safety standards against which new hardware modules are measured. The safety delivered by a proof-carrying bitstream is as strong (or weak) as its specification. The producer of a hardware module cannot adhere to any standard that is not formally described in the safety policy. The concept of proof-carrying hardware cannot help a designer to detect what security constitutes in a particular environment. This problem is inherent to all security challenges: only known threats can be eliminated.

There are security and safety risks, though, that are not covered by proof-carrying hardware due to its inherent limitations: Proof-carrying hardware aims at establishing trust in hardware modules from untrusted sources to guarantee that said hardware modules do not compromise the integrity of the target platform running the reconfigurable device.

Proof-carrying hardware is not concerned with the prevention of physical attacks, e.g. cloning or denial-of-service attacks as described in Section 2.2.1.

Another inherent limitation of the proof-carrying hardware principle lies in its goal to shift the workload to the producer, particularly regarding the effort for the verification, i.e. proof computation. For this, a proof-carrying bitstream contains information that helps the consumer to establish trust in certain security features. The validation of that proof should be faster than the elaborate formal or otherwise verification of said security feature. Hence, verification problems that are particularly suitable for the proof-carrying hardware concept are those that offer a faster validation than verification. An example of a verification problem that in its current state does not have a fast way of validating a proof is the detection of short-circuits as suggested by Beckhoff et al. in [10]. A bitstream scanner, which is equipped with a build-in algorithm, detects long-term short-circuits. The producer can use that technique to screen the bitstream but has no means to communicate the absence of short-circuit configurations among the multiplexers to the consumer. The consumer has to inspect each part of the circuit, bitstream or graph representation of the circuit again as no proof can be carried to the consumer and no workload can be shifted towards to producer. The consumer would therefore not benefit from proof-carrying hardware as no shift of workload can occur.

6.2 Usability of Combinational Equivalence Checks

In Section 3.3, I identified the usability as a vital aspect of the proof-carrying hardware concept. As defined, the usability of proof-carrying hardware means a quick and unconsuming validation of the safety proof, especially in comparison to the overall cost of security verification and validation. Hence, a shift of the workload towards the producer is of the essence.

All measurements in this chapter were taken on a Linux 2.6 machine with an Intel Xeon 2.40GHz CPU and 4 GB RAM, see [30, 27, 28]. For all measurements p_i of the producer's tools and measurements c_j of the consumer's tools regarding either runtime or memory usage, the total workload is $workload = \sum_i(p_i) + \sum_j(c_j)$.

A first indication for the potential of this novel concept is the runtime comparison of the verification and validation of cnf functions, i.e. the computation of the proof and the checking of the proof, respectively. Table 6.1 compares the runtime of the SAT solver (PicoSAT) and the proof checker (TraceCheck) for cnf problems from the 2008 SAT-Race_TS_1 benchmark. PicoSAT proves the unsatisfiability of the cnf formula, which is an NP-complete problem. TraceCheck validates the resolution proof by checking every resolution step, a task whose complexity is linear in the number of resolution steps and constant in the number of literals of the original cnf. For a successful shift or workload, the difference in complexity has to determine the runtimes for the test files.

The cnf test functions differ greatly in the number of variables and clauses. The last column of Table 6.1 shows that the effort for computing the proof with PicoSAT makes up for 74.33% to 99.70% percent of the total workload, composed of computing the proof trace with PicoSAT and validating the proof by computing a resolution trace with TraceCheck.

cnf instance	Size of cnf [Vars	Clauses]	TraceCheck total [s]	PicoSAT [s]	workload [%]
een-tipb-sr06-par1.cnf	163647	484827	0.052	6.772	99.24
een-tipb-sr06-tc6b.cnf	40196	115775	0.044	3.34	98.70
goldb-heqc-desmul.cnf	28902	179895	4.357	80.876	94.89
goldb-heqc-rotmul.cnf	5980	35229	7.459	37.368	83.36
hoons-vbmc-s04-05.cnf	8503	25097	5.217	15.312	74.59
hoons-vbmc-s04-07.cnf	25900	77627	33.866	158.698	82.41
manol-pipe-c10b.cnf	43517	129265	19.583	247.741	92.67
manol-pipe-c10ni_s.cnf	204664	609478	0.046	6.447	99.29
manol-pipe-c6id.cnf	82022	242044	3.53	91.787	96.30
manol-pipe-c6n.cnf	37147	110077	3.065	53.91	94.62
manol-pipe-c6nid_s.cnf	148051	438562	0.062	7.772	99.21
manol-pipe-c7_i.cnf	13023	38509	0.891	22.162	96.13
manol-pipe-c7idw.cnf	112620	333058	4.529	129.983	96.63
manol-pipe-c8b_i.cnf	14052	41596	6.977	76.602	91.65
manol-pipe-c8_i.cnf	32057	95005	0.827	16.892	95.33
manol-pipe-c8n.cnf	53697	159595	6.903	108.065	94.00
manol-pipe-f6b.cnf	37002	109570	0.747	7.29	90.71
manol-pipe-f6n.cnf	37452	110920	0.997	8.21	89.17
manol-pipe-g10idw.cnf	174122	516784	8.455	141.273	94.35
manol-pipe-g6bid.cnf	40371	118192	0.474	6.141	92.83
manol-pipe-g7n.cnf	23936	70492	0.968	7.009	87.87
narai-vpn-10s.cnf	2270930	8901946	0.775	255.04	99.70
schup-l2s-s04-abp4.cnf	14809	48429	30.332	106.983	77.91
velev-npe-1.0-02.cnf	3295	35407	13.592	42.004	75.55
velev-sss-1.0-cl.cnf	1453	12526	12.431	35.988	74.33

Table 6.1: Runtime comparison of PicoSAT and TraceCheck for benchmarks of the 2008 SAT-Race_TS_1. Given are the complexity and length of the test function, the runtime for validation (TraceCheck) invested by the consumer, the runtime for verification (PicoSAT) invested by the consumer, and the percentage of the workload performed by the producer.

This first evaluation demonstrates the value for the consumer of shifting the workload away from a reconfigurable system and instead towards the producer of a hardware module. The gain for the consumer may vary, depending on the size and complexity of the proof, but is always noticeable for these test cases. Figure 6.2 displays how the total runtime and runtime for PicoSAT are in most cases indistinguishable.

Table 6.1 also indicates the following conclusions: The complexity and length of a cnf formula seems to influence the shift of workload, i.e. the percentage of the total runtime performed by the producer. The function hoons-vbmc-s04-05.cnf contains 8503 variables

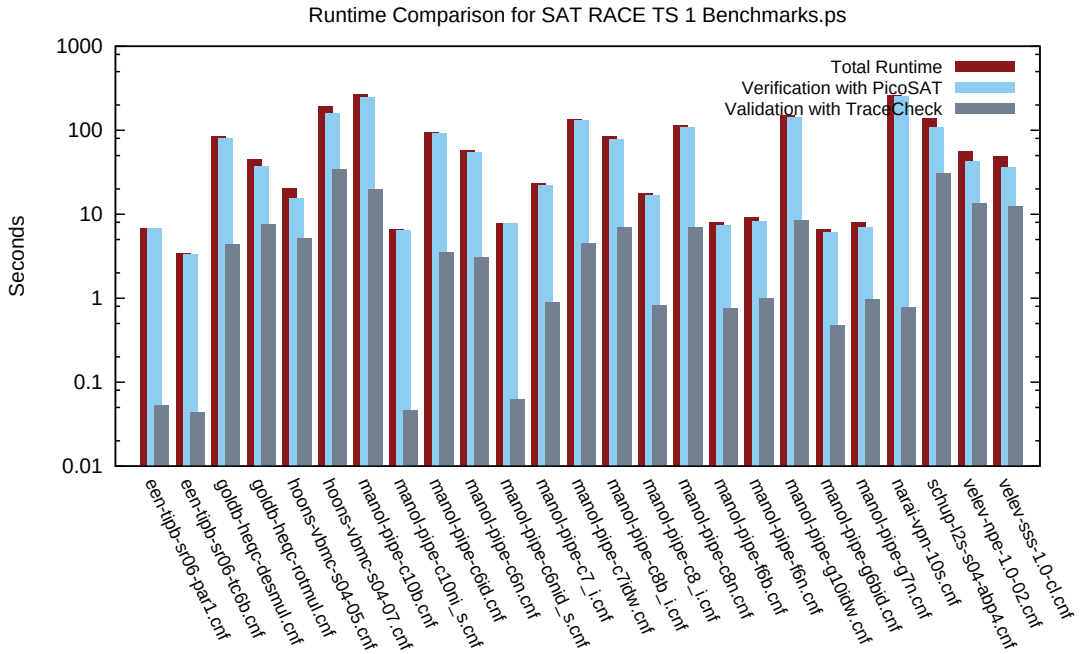


Figure 6.2: Runtime comparison for verification (PicoSAT) and validation (TraceCheck) of unsatisfiable 2008 SAT-Race_TS_1 test cases. (logarithmic scale)

and consists of 25097 clauses, the producer's runtime is higher by a factor of 2.94 than the consumer's and makes up 74.59% of the workload. Functions manol-pipe-c8_i.cnf is listed with 32057 variables and 95005 clauses, which is 3.7 times more than the previous functions. This results in a factor 20.43 and a shift of workload of 95.33%. The largest test function is narai-vpn-10s.cnf with 2270930 variables and 8901946 clauses. The shift of workload is 99.70%. This is encouraging as it indicates that the principle of a light workload for the consumer of a newly delivered hardware module, which is essential to proof-carrying hardware, can be realized.

To further examine the possible shift of workload, I set up a preliminary tool flow shown in Figure 6.3 to take more steps of the overall work flow of the production process and the verification process into consideration. This proof of concept tool flow is a simplified version of the scenario shown in Figure 5.2 as it uses ODIN II's predecessor ODIN and the simple ComPose tool which produces a concatenation of the single text files to simulate a bitstream. The preliminary tool flow serves to demonstrate the feasibility of runtime combinational equivalence checks as an application of proof-carrying hardware and to validate whether it is possible to shift the verification workload from the consumer to the producer. It is neither necessary nor the intention of this preliminary investigation to verify all steps of the production, e.g., FPGA back-end synthesis tools, to check for

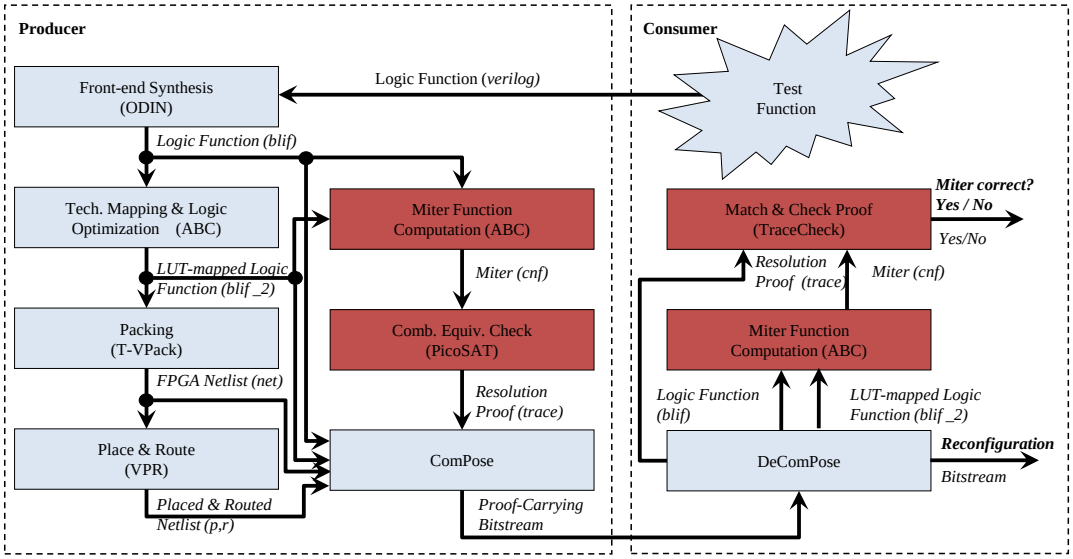


Figure 6.3: Preliminary tool flow for CEC testing environment.

test function	type
converter	letter subset of EBCDIC to ASCII
parity function	128-bit
n-bit adder / subtractor	$n = 8, 16, 32, 64$
n-bit unsigned multiplier	$n = 6, 8, 10, 16, 32, 64$

Table 6.2: List of the 12 test functions for runtime and memory measurements of preliminary tool flow.

correct transmission or to completely implement the consumer's functions.

I report on results using the test functions listed in Table 6.2. I have chosen these test functions and their according input size variations to contrast functions which are presumably rather easy to verify with more demanding ones, i.e. the multipliers. As mentioned before, an important metric for the feasibility of proof-carrying hardware, in this case CEC, is the required computation time, especially for the consumer. I measure the runtime of all tools in the preliminary tool flow. On the consumer side, ABC' only recomputes the miter without performing logic optimization.

Table 6.3 presents the runtime measurements for the test functions. The first row gives the computation time for the producer's side of the tool flow while the second row depicts the runtimes for the consumer. The table clearly shows that the results differ greatly for the different types of test functions as the multiplier test functions form a separate group. First, with growing number of inputs multipliers synthesized from LUTs become huge functions which is demonstrated by the high runtimes for VPR. Second, SAT solvers

	Producer [s]					
Test Function	Odin	ABC	T-VPack	VPR	PicoSAT	ComPose
converter	0.187	0.348	0.005	2.040	0.008	0.080
128-bit parity	0.194	0.350	0.100	10.892	0.036	0.111
8-bit add/sub	0.183	0.273	0.007	2.126	0.015	0.034
16-bit add/sub	0.122	0.279	0.011	5.520	0.048	0.074
32-bit add/sub	0.186	0.330	0.012	14.428	0.099	0.154
64-bit add/sub	0.195	0.432	0.220	40.674	0.244	0.327
6-bit multiplier	0.179	0.332	0.008	5.304	0.540	0.145
8-bit multiplier	0.184	0.450	0.010	13.205	15.337	1.148
10-bit multiplier	0.183	0.645	0.014	26.589	256.119	18.849
16-bit multiplier	0.205	1.473	0.040	133.814	3732.031*	
32-bit multiplier	0.229	6.236	0.163	2116.210	3895.797*	
64-bit multiplier	0.527	27.424	0.726	36447.768	6387.947*	

	Consumer [s]				Producer	
Test Function	DeComP.	ABC'	TraceCheck	total	total	workload
converter	0.004	0.148	0.013	0.165	2.668	93%
128-bit parity	0.007	0.156	0.027	0.190	11.683	98%
8-bit add/sub	0.004	0.201	0.010	0.215	2.638	92%
16-bit add/sub	0.004	0.206	0.035	0.245	6.054	96%
32-bit add/sub	0.009	0.215	0.059	0.283	15.209	98%
64-bit add/sub	0.015	0.234	0.126	0.375	42.092	99%
6-bit multiplier	0.009	0.234	0.483	0.726	6.508	89%
8-bit multiplier	0.123	0.213	11.179	11.515	30.334	72%
10-bit multiplier	1.807	0.214	190.630	192.651	302.339	61%
16-bit multiplier						
32-bit multiplier						
64-bit multiplier						

Table 6.3: Runtime measurements for producer and consumer in the CEC scenario. Measurements aborted due to a lack of memory are marked with an asterisk.

have difficulties in proving the unsatisfiability of multiplier miters which is reflected by the PicoSAT runtimes. In fact, in the experiments, PicoSAT aborted the computation due to a lack of memory for multipliers with $n = 16$ and higher. Consequently, I could not conduct measurements for the consumer side of the tool flow for these functions. I mark these cases with an asterisk in Table 6.3.

The main observation from Table 6.3 is the difference in time effort between producer and consumer. ABC' is less costly than ABC with the gap widening for the more complex

test functions. Most importantly, with one exception, there is a notable difference between the PicoSAT and TraceCheck runtimes, even if not as pronounced as expected. This might be due to the fact that on one hand unsatisfiability is easily proven for the miters built from these simple test functions. Also, the generated netlists are quite large which means all tools processing netlists spend substantial time in file I/O. For the more complex test functions the SAT solver dominates the producer's overall runtime, an effect that can be seen in going from the 8-bit to the 10-bit multiplier. With Table 6.3, I also give an overview of the total runtimes for both the consumer and producer side. The table also reports the producer's percentage of the total workload consisting of both the consumer's and producer's runtime. The data shows that I succeeded in my attempt to shift the majority of the workload from the consumer platform to an external resource.

Table 6.4 gives the results for the memory usage for each tool and test function. I measured the peak memory usage with Valgrind [61], i.e. the massif tool, and included stacks as well as heaps in the measurements. The first row displays the memory usage on producer side, the second row show the measurement results on consumer side. As in Table 6.3, the numbers marked with an asterisk represent a minimum value, measured before the tool aborted the computation as it ran out of memory.

I note that PicoSAT and TraceCheck, followed by VPR and ABC, are the most memory consuming tools on the producer's and consumer's side, respectively. Formal verification is therefore among the most memory demanding tasks in the scenario. The larger multiplier test functions with 16-bit inputs or more caused PicoSAT to run out of memory. Comparing TraceCheck and PicoSAT, the consumer has to invest slightly more in memory than the producer. As opposed to the runtimes, in which case the producer is able to burden a major part of the total workload, the memory resource requirement is still considerable for the consumer in this current prototype tool flow. I attribute this to the file I/O since TraceCheck not only reads the proof trace but also writes the resolution trace as output.

6.3 Usability of Memory Access Monitor Verification

For a more conclusive answer with regard to the runtime workload shift, I used the prototype tool flow as shown in Figure 5.2 with reference monitor circuits described in Section 4.3 as test functions. For this purpose I combined a set of test functions, including all previously elaborated types of reference monitors, see Table 6.5.

The test functions offer different levels of complexity to give a more meaningful range of measurements. To gain some perspective on the complexity of the different policy instances, i.e. test functions, I elaborate and compare two different test functions. The Iso4 function is an instance of the Isolation model, a static memory access policy introduced in Section 4.3.2. Table 6.6 shows the actual policy instance as it was compiled into a Verilog file and processed in the tool flow. Figure 6.4 gives an overview of the access rights according to modules and ranges. As the policy is a static one, the access rights are static and cannot change at runtime. Therefore, there is only a single state for the reference monitor plus an additional error state in case of access rights violations. Even with significantly more ranges, compartment, and modules, this reference monitor barely

	Producer					
Test Function	Odin	ABC	T-VPack	VPR	PicoSAT	ComPose
converter	0.997 MiB	13.79 MiB	239.6 KiB	1.207 MiB	69.42 KiB	62.17 KiB
128-bit parity	935.8 KiB	13.60 MiB	310.1 KiB	3.465 MiB	189.2 KiB	61.78 KiB
8-bit add/sub	711.4 KiB	13.90 MiB	243.5 KiB	1.218 MiB	87.71 KiB	61.62 KiB
6-bit add/sub	771.4 KiB	14.05 MiB	267.4 KiB	2.306 MiB	238.8 KiB	61.62 KiB
32-bit add/sub	899.6 KiB	14.27 MiB	315.4 KiB	4.480 MiB	488.5 KiB	61.62 KiB
64-bit add/sub	1.130 MiB	14.58 MiB	533.8 KiB	8.341 MiB	1.010 MiB	61.62 KiB
6-bit multiplier	774.6 KiB	13.93 MiB	263.6 KiB	2.298 MiB	1.380 MiB	62.70 KiB
8-bit multiplier	803.9 KiB	13.95 MiB	301.1 KiB	4.086 MiB	25.74 MiB	64.28 KiB
10-bit multiplier	887.3 KiB	14.10 MiB	391.9 KiB	6.425 MiB	374.8 MiB	71.20 KiB
16-bit multiplier	1.201 MiB	14.42 MiB	937.0 KiB	16.46 MiB	2.477 GiB*	
32-bit multiplier	2.750 MiB	19.83 MiB	3.590 MiB	67.33 MiB	2.742 GiB*	
64-bit multiplier	8.858 MiB	48.98 MiB	14.41 MiB	271.0 MiB	2.699 GiB*	

	Consumer		
Test Function	DeComP.	ABC'	TraceCheck
converter	62.12 KiB	10.06 MiB	81.00 KiB
128-bit parity	61.73 KiB	9.937 MiB	181.1 KiB
8-bit add/sub	61.58 KiB	10.54 MiB	101.1 KiB
16-bit add/sub	61.58 KiB	10.70 MiB	268.4 KiB
32-bit add/sub	61.58 KiB	10.98 MiB	485.9 KiB
64-bit add/sub	61.58 KiB	11.69 MiB	1.038 KiB
6-bit multiplier	62.66 KiB	10.38 MiB	2.167 MiB
8-bit multiplier	64.23 KiB	10.57 MiB	43.97 MiB
10-bit multiplier	71.15 KiB	10.87 MiB	652.8 MiB
16-bit multiplier			
32-bit multiplier			
64-bit multiplier			

Table 6.4: Memory usage measurements for producer and consumer in the CEC scenario. Measurements aborted due to a lack of memory are marked with an asterisk.

increases in complexity.

An instance of the Chinese Wall model is test function Chin1, as shown in Table 6.7. The instance features two conflict-of-interest (COI) classes, each containing two ranges. The subject Module_1 can choose to access one member of each class. The actual complexity of the dynamic policy is shown in Figure 6.5: The state machine resulting from this access scenario operates with 10 states, the 9 squares represent the states for memory access, the error state is not shown. Each memory access is a state transition and leaves less

test function	ranges	modules	security levels	states
Biba1 / BL1 / High1 / Low1	2	2	2	2 / 2 / 3 / 3
Biba2 / BL2 / High2 / Low2	2	3	3	2 / 2 / 3 / 3
Biba3 / BL3 / High3 / Low3	3	3	3	2 / 2 / 7 / 7
Biba4 / BL4 / High4 / Low4	2	2	4	2 / 2 / 3 / 3
Biba5 / BL5 / High5 / Low5	2	4	4	2 / 2 / 7 / 5
Biba6 / BL6 / High6 / Low6	4	4	4	2 / 2 / 7 / 7
test function	ranges	modules	compartments	states
Iso1	2	2	2	2
Iso2	4	2	2	4
Iso3	16	8	4	32
Iso4	16	16	4	64
test function	ranges	modules	COI classes	states
Chin1	4	1	2	10
Chin2	6	1	2	17
Chin3	6	1	3	28
Chin4	8	1	4	82

Table 6.5: List of the 32 reference monitor test functions based on the six types of memory access policies introduced in Section 4.3.

options for further memory accesses. Any further memory range or COI class dramatically increases the complexity of the resulting reference monitor and its state machine. Test function Chin4 manages the access to 8 memory ranges in 4 COI classes and requires a state machine with 82 states to perform this task. For an overview of the required states in the state machine refer to Table 6.5.

The increase in complexity is different for the varying reference monitors, the set of test functions offers therefore an adequate variation in complexity.

6.3.1 Computational Effort

As a first step, I consider again only the runtime for PicoSAT and TraceCheck, i.e. verification and validation. Figure 6.6 shows that the verification of the miter function is always less than the validation of the resolution proof trace. In case of the very complex combinational miters, i.e. Iso3 and Iso4, I notice that a) verification and validation both are more costly and b) the difference between verification and validation grows exponentially. The runtime for both, PicoSAT and TraceCheck, is in these cases almost indistinguishable from the runtime of PicoSAT itself. This means that the consumer is left with only a fraction of the overall cost of security.

Table 6.8 lists the runtimes for PicoSAT and TraceCheck as well as the percentage of

Isolation;					
$Compartment_1$	$\longrightarrow$	$Module_1$;	$Compartment_3$	$\longrightarrow$	$Module_9$;
$Compartment_1$	$\longrightarrow$	$Module_2$;	$Compartment_3$	$\longrightarrow$	$Module_{10}$;
$Compartment_1$	$\longrightarrow$	$Module_3$;	$Compartment_3$	$\longrightarrow$	$Module_{11}$;
$Compartment_1$	$\longrightarrow$	$Module_4$;	$Compartment_3$	$\longrightarrow$	$Module_{12}$;
$Compartment_1$	$\longrightarrow$	$Range_1$;	$Compartment_3$	$\longrightarrow$	$Range_9$;
$Compartment_1$	$\longrightarrow$	$Range_2$;	$Compartment_3$	$\longrightarrow$	$Range_{10}$;
$Compartment_1$	$\longrightarrow$	$Range_3$;	$Compartment_3$	$\longrightarrow$	$Range_{11}$;
$Compartment_1$	$\longrightarrow$	$Range_4$;	$Compartment_3$	$\longrightarrow$	$Range_{12}$;
$Compartment_2$	$\longrightarrow$	$Module_5$;	$Compartment_4$	$\longrightarrow$	$Module_{13}$;
$Compartment_2$	$\longrightarrow$	$Module_6$;	$Compartment_4$	$\longrightarrow$	$Module_{14}$;
$Compartment_2$	$\longrightarrow$	$Module_7$;	$Compartment_4$	$\longrightarrow$	$Module_{15}$;
$Compartment_2$	$\longrightarrow$	$Module_8$;	$Compartment_4$	$\longrightarrow$	$Module_{16}$;
$Compartment_2$	$\longrightarrow$	$Range_5$;	$Compartment_4$	$\longrightarrow$	$Range_{13}$;
$Compartment_2$	$\longrightarrow$	$Range_6$;	$Compartment_4$	$\longrightarrow$	$Range_{14}$;
$Compartment_2$	$\longrightarrow$	$Range_7$;	$Compartment_4$	$\longrightarrow$	$Range_{15}$;
$Compartment_2$	$\longrightarrow$	$Range_8$;	$Compartment_4$	$\longrightarrow$	$Range_{16}$;

Table 6.6: Instance of the Isolation Model used as test function Iso4, including 4 compartments and 16 memory ranges and modules. Each compartment contains 4 ranges and 4 modules that have access to the according ranges.

Chinese;		
$Class_1$	$\longrightarrow$	$Range_1$;
$Class_1$	$\longrightarrow$	$Range_2$;
$Class_2$	$\longrightarrow$	$Range_3$;
$Class_2$	$\longrightarrow$	$Range_4$;
Subject	$\longrightarrow$	$Module_1$;

Table 6.7: Instance of the Chinese Wall Model used as test function Chin1.

the overall workload for the test function that PicoSAT constitutes. The table illustrates two facts:

First of all, reference monitors based on dynamic memory access policies, i.e. the bottom three types, offer a greater shift of workload than those based on static ones. The average shift of workload of all Biba test functions is a mere 61.7%, and for Bell & LaPadula and Isolation these numbers are 63,04%, and 80.52%, respectively. I attribute this to the small size and lack of complexity of the miter functions. The absolute runtime in seconds is low enough to suggest that it consists almost exclusively of the file I/O overhead, the actual computational effort seems to be insignificant. In contrast, for the Low Watermark functions, the shift of workload is 98.9%. For High Watermark and Chinese Wall reference monitors, PicoSAT performs a respective 98.61% and 99.08% of the verification and validation workload. For these test cases, the computational effort is

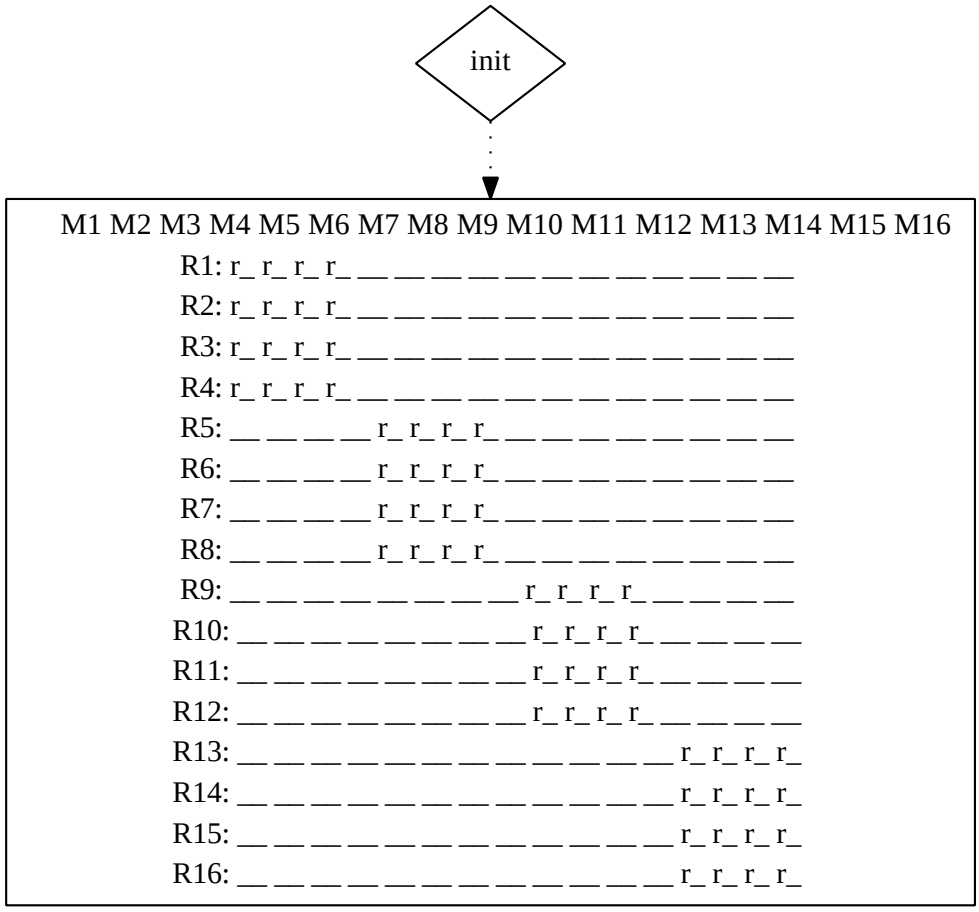


Figure 6.4: Static memory access rights of the Iso4 test function, an instance of the Isolation memory access policy. Each block of $_r$s in the matrix visualizes one compartment.

by far greater than any overhead due to file handling.

The second noticeable fact is that the overall shift of workload for all 32 test instances accumulates to 98.86%. That indicates that the gain from larger functions outweighs the rather low shift of workload occurring for smaller and less complex functions.

The next analysis considers the overall runtime of the tool flow with regard to its distribution between the consumer and producer. Figure 6.7 and Table 6.9 display the consumer's and the producer's overall runtime for each test function. Refer to Figure 5.2 to see the list of included tools and work steps. I make the following observations:

- Firstly, for the consumer, the differences within the set of combinational test functions is rather negligible. For the producer, the differences in runtime for the first

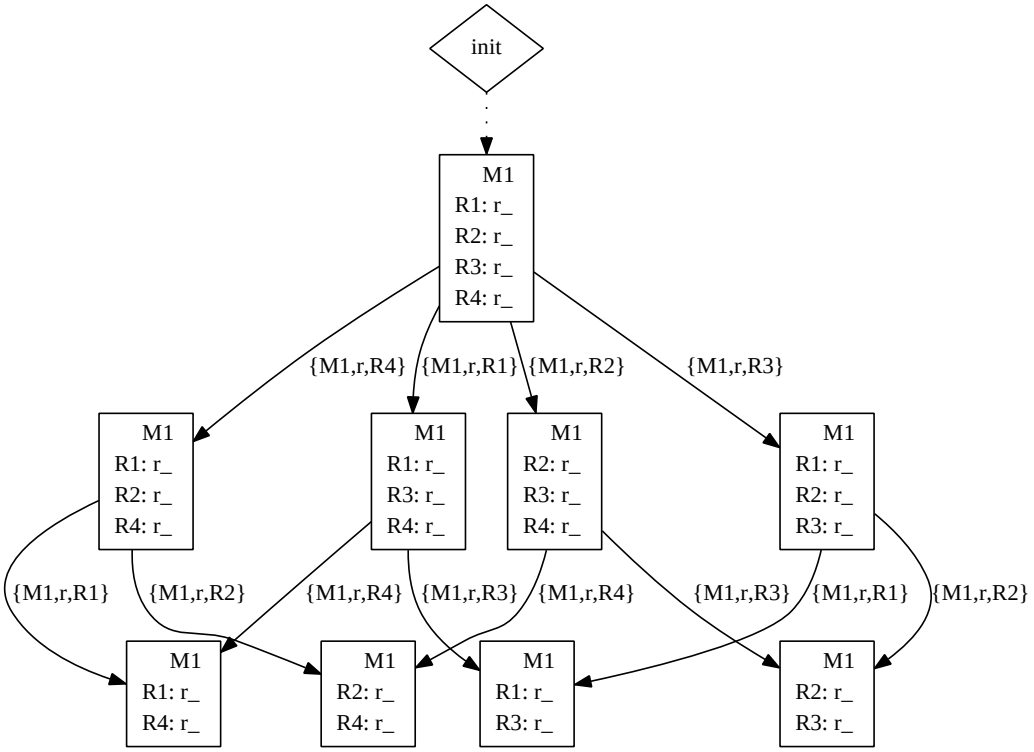


Figure 6.5: Dynamic memory access rights of the Chin1 test function, an instance of the Chinese Wall memory access policy. Module M_1 starts with four choices for memory access. Each access (denoted by a triple of module id, access type, and memory range at the arrows) eliminates access to any other memory range from the same COI class until only two memory ranges from different COI classes are available for access.

(combinational) half of the test functions are less pronounced than they are within the second (sequential) half of the test functions. As presented in Table 6.9, only the test functions Iso3 and Iso4 show a slight rise in runtime cost, particularly for the producer with respective runtimes of 3.53 s and 5.021 s. For the second half of the test set, the consumer runtime ranges from 1.213 s to 26.270 s, the producer runtime ranges from 2.114 s to 130.522 s. Those wider ranges reflect the differences in complexity of the various test functions.

- Secondly, for both, the consumer and the producer, the sequential test functions are exponentially more costly than the combinational ones. With the exception of Iso3 and Iso4 on the producer's toll flow side, all reference monitor test functions based on static memory access policies have a lower runtime than those based on dynamic policies. In fact, 96.94% of the consumer's and 90.83% of the producer's

total workload falls towards the second half of the test functions.

- Thirdly, the shift of workload towards the producer is higher for the first half of test functions, i.e. the less complex ones. Despite the fact that the shift of workload scaled well for the verification as demonstrated in Table 6.8, it did not correlate with the complexity of the test function for the complete tool flow including the verification. For the first three types of test function, the Biba, BL, and Iso, the producer was burdened with 88.25%, 88.15%, and 94.36%, respective, of the total runtime. For the Low, High, and Chin test functions, these percentages are 65.22%, 65.34%, and 80.11%. This can be attributed to the recomputation of the miter function on the consumer's side of the tool flow, see Figure 5.2. As Figure 6.7 shows, the computation of the miter makes up almost all of the consumer's workload. I elaborated in Section 6.1 that the consumer performs a recomputation of the miter to assure the match between design specification, hardware module, and safety proof. This finding encourages the search for an optimized and less costly way of assuring said match to further realize the shift of workload which is essential to the principle proof-carrying hardware.

- The last and possibly most important result of the tool flow runtime comparison is that for every test instance, the producer deals with the majority of the workload. The consumer is again only left with a fraction of the total runtime. For all 32 test cases, the producer performs 76.98% of the total workload. This clearly indicates the great potential of proof-carrying hardware.

Combining all findings of this section, I can conclude the following: The shift of workload to the producer's side did succeed. Especially the shift of the security workload, i.e. verification and validation, is encouraging, as it scaled with the test functions' complexity: over 98% of the total workload for all 32 test cases is performed by the producer. This is due to the fact that the groups of complex test cases such as Chin, with an average shift of workload of 99.08%, outweighs the less complex groups like Biba which only feature a group average of 61.7%. For the overall workload, including all synthesis and verification steps, the producer had exponentially more runtime than the consumer in all cases. For the first half of test functions, the percentage of the overall workload shifted to the producer is 85% - 96%. For the second half of the test functions, 62% - 83% of the workload falls to the producer, see Table 6.9. For all 32 test cases, the producer performs an average 76.98% of the entire workload.

6.3.2 Memory Requirement

Another aspect of the usability which proof-carrying hardware has to provide is a light memory usage burden for the consumer of hardware modules. The following analysis provides a comparison of the memory usage for tools used by the consumer and producer, respectively.

As a first analysis, I consider the memory usage difference between the verification and validation, i.e. PicoSAT and TraceCheck, see Figure 6.8 and Table 6.10. For the combina-

Test unction	TraceCheck	PicoSAT	
	total [s]	total [s]	workload [%]
Biba1	0.003	0.004	57.14
Biba2	0.002	0.003	60.00
Biba3	0.004	0.006	60.00
Biba4	0.002	0.003	60.00
Biba5	0.003	0.005	62.50
Biba6	0.004	0.008	66.67
BL1	0.002	0.003	60.00
BL2	0.003	0.003	50.00
BL3	0.003	0.006	66.67
BL4	0.002	0.002	50.00
BL5	0.003	0.005	62.50
BL6	0.004	0.010	71.43
Iso1	0.001	0.002	66.67
Iso2	0.002	0.005	71.43
Iso3	0.016	0.075	82.42
Iso4	0.033	0.133	80.12
Low1	0.016	0.387	96.03
Low2	0.018	0.456	96.20
Low3	0.059	4.371	98.67
Low4	0.017	0.361	95.50
Low5	0.031	7.547	99.59
Low6	0.060	4.996	98.81
High1	0.016	0.365	95.80
High2	0.017	0.460	96.44
High3	0.058	4.176	98.63
High4	0.016	0.369	95.84
High5	0.033	3.680	99.11
High6	0.060	5.102	98.84
Chin1	0.032	2.969	98.93
Chin2	0.060	1.406	95.91
Chin3	0.116	11.177	98.97
Chin4	0.254	34.189	99.26

Table 6.8: Runtime comparison of PicoSAT (SAT) and TraceCheck (Check) for reference monitor test functions as listed in Table 6.5.

Test Function	Consumer	Producer	
	total [s]	total [s]	workload [%]
Biba1	0.141	0.903	86.49
Biba2	0.132	0.903	87.25
Biba3	0.141	1.076	88.41
Biba4	0.135	0.818	85.83
Biba5	0.139	1.020	88.01
Biba6	0.136	1.471	91.54
BL1	0.132	0.885	87.02
BL2	0.133	0.839	86.32
BL3	0.131	1.021	88.63
BL4	0.130	0.798	85.99
BL5	0.136	1.015	88.18
BL6	0.134	1.364	91.05
Iso1	0.130	0.846	86.68
Iso2	0.131	0.977	88.18
Iso3	0.164	3.530	95.56
Iso4	0.195	5.021	96.26
Low1	1.270	2.200	63.40
Low2	1.386	2.313	62.53
Low3	3.895	7.710	66.44
Low4	1.243	2.136	63.21
Low5	2.432	4.459	64.71
Low6	4.350	8.519	66.20
High1	1.213	2.138	63.80
High2	1.426	2.502	63.70
High3	3.814	6.983	64.68
High4	1.216	2.114	63.48
High5	3.092	5.950	65.80
High6	4.319	8.744	66.94
Chin1	2.659	4.864	64.66
Chin2	4.203	7.764	64.88
Chin3	8.358	23.962	74.14
Chin4	26.270	130.522	83.25

Table 6.9: Runtime comparison of producer and consumer for reference monitor test functions as listed in Table 6.5.

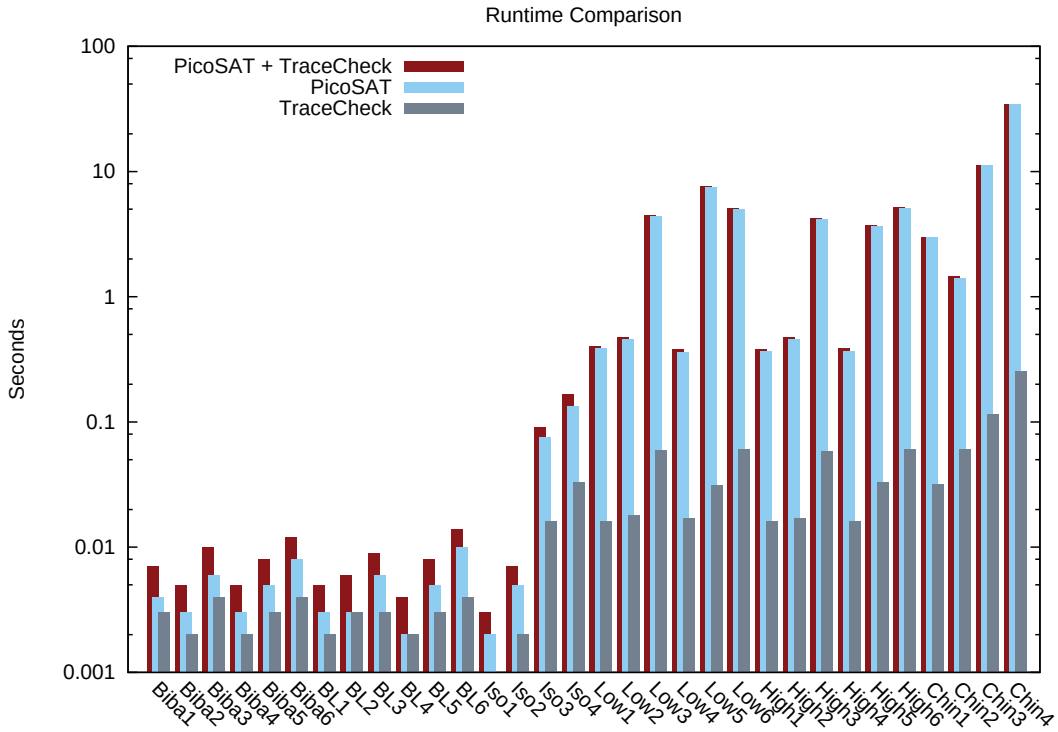


Figure 6.6: Runtime Comparison of verification vs. validation of the Table 6.5 reference monitor test functions. (logarithmic scale)

tional miters, verification and validation memory footprints are rather indistinguishable. For the bounded sequential equivalence checks, i.e. the second half of the test functions, PicoSAT exceeds TraceCheck exponentially. Also, the difference between the first half and second half of the test set, i.e. the reference monitors based on static and those based on dynamic memory access policies, is exponential for the consumer and producer. This seems to indicate that the file size of the miter functions is a rather dominant factor for the combinational verification and validation. For the bounded sequential test cases, the memory usage is dominated more by the actual computation than the file sizes. Hence, the verification becomes exponentially more memory-intense than the validation. For the three dynamic types of test functions, the producer burdens 84.39%, 84.44%, and 86.19% of the workload for verification and validation. Since the workload for the first three test types, Biba, BL, and Iso, is very low in comparison with about 49%, 85.37% of the overall memory usage falls to the producer.

The second measurement considers the cumulative overall memory usage for the consumer and producer when executing all tasks in the tool flow, see Figure 6.9 and Table 6.11.

Firstly, for the consumer and producer, the differences within the set of combinational test functions is rather negligible. As presented in Table 6.11, only the test functions Iso3 and Iso4 show a very slight rise in memory usage. For the second half of the test set,

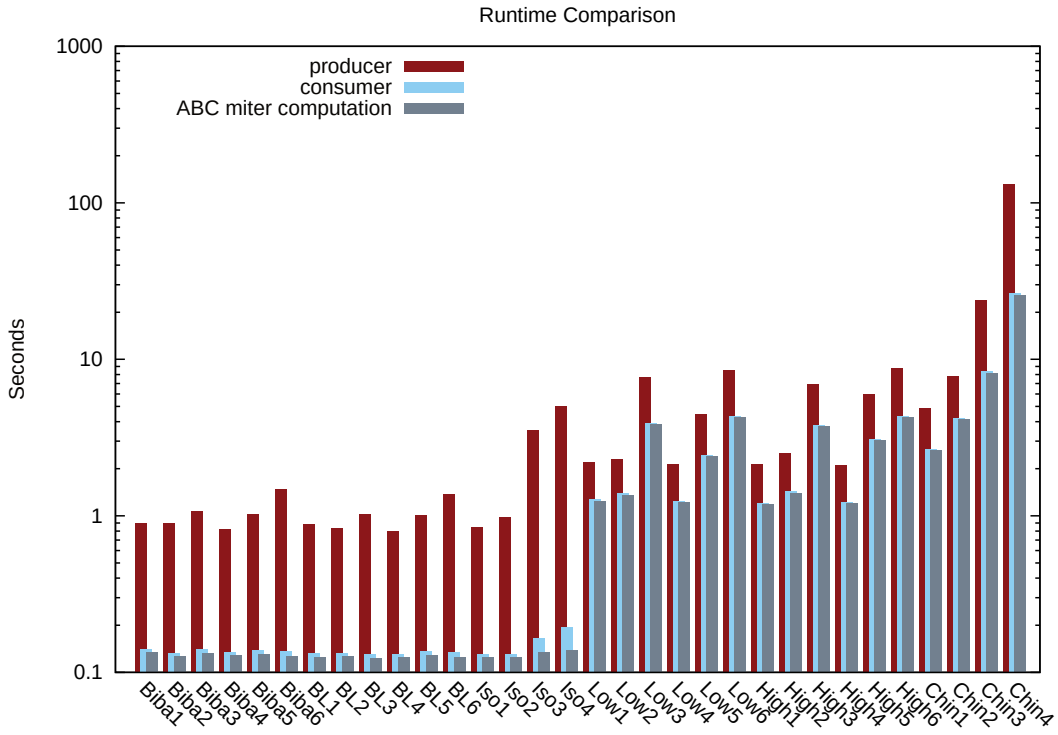


Figure 6.7: Runtime Comparison of producer tool flow vs. consumer tool flow for Table 6.5 reference monitor test functions. (logarithmic scale)

the consumer memory usage ranges from 15474.89 KiB to 1996774.40 KiB, the producer memory usage ranges from 39481.84 KiB to 1928449.40 KiB. Those wider ranges reflect again, as happened for the runtime comparison in Table 6.9, the differences in complexity of the various test functions.

Secondly, for both, the consumer and the producer, the sequential test functions are exponentially more costly than the combinational ones. All reference monitor test functions based on static memory access policies have a lower memory footprint than those based on dynamic policies. Similar to the results received for the runtime analysis, 96.10% of the consumer's and 90.63% of the producer's total workload falls towards the second half of the test functions.

Thirdly, the shift of workload towards the producer is higher for the first half of test functions, i.e. the less complex ones. The shift of workload for the first three types of test functions, the Biba, BL, and Iso accumulates to 71.86%, 71.86%, and 72.35%, respectively. The lower number for the last three types of functions are 51.08%, 51.04%, and 49.43%, for Low, High, and Chin. Figure 6.9 shows how the consumer's memory usage is very close to the memory usage required for the miter recomputation with the ABC tool. Hence, I attribute it to the miter recomputation that the overall shift of the workload, i.e. memory usage, for all 32 test functions is a mere 51.66%. As pointed out during the runtime

comparison analysis in the previous section, a better way of matching the resolution proof to the design specification in the tool flow, see Figure 5.2, would enhance the desired shift of workload.

To summarize the findings in this section, I conclude: The memory usage comparison is somewhat similar to the runtime comparison. The computation of the resolution proof is exponentially more costly than validating the proof. Also, the test functions based on dynamic memory access policies are exponentially more costly than those based on static policies. A conclusion also drawn in Section 6.3.1.

The results have also shown that the memory usage for the consumer is largely dominated by the miter recomputation. Again, a result known from Section 6.3.1. This further manifests the need for a less costly way of matching the safety proof to its design specification and bitstream to allow for a lighter workload for the proof-carrying bitstream consumer. This would also increase the shift of workload, in this case the memory usage, which is 51.66% of the total workload for all 32 test cases. It would be desirable to further the advantages of proof-carrying hardware to achieve an overall shift of workload that is similar to the 85.37% shift of workload that had been achieved for the mere verification and validation of the safety, i.e. proof computation and proof validation, which shows the potential of the usability of proof-carrying hardware with regard to memory footprints.

6.4 Chapter Conclusion

In this chapter, I performed an experimental evaluation of the proof-carrying hardware prototype. The proof-carrying hardware approach, as elaborated in Chapter 3, needs to provide robustness and usability. I examined the robustness in Section 6.1. For the prototype tool flow, I discussed how possible malicious or unintentional manipulation of the data will eventually lead to validation failure, thus alarming the consumer of the proof-carrying bitstream. Proof-carrying hardware succeeds in establishing trust in a newly delivered hardware module from an untrusted source.

The usability of this novel concept is expressed in the shift of workload. The shift of workload expresses the distribution of the workload between the producer, who should provide as much as possible, and the consumer, who should be burdened as little as possible. I investigated the shift of workload from producer to consumer concerning the runtime and memory usage for the verification and validation process and also the entire prototype tool flow. The test functions are miter cnfs based on reference monitors for static and dynamic memory access policies, see Chapter 4. The results can be summarized as follows and hold true for the runtime as well as memory usage:

- The dynamic memory access policies result in more complex test functions which are more costly to verify and validate.

- The more costly functions show a greater shift of workload for the verification and validation process.

Test Function	TraceCheck	PicoSAT	
	max [KiB]	max [KiB]	workload [%]
Biba1	61.98	58.73	48.65
Biba2	62.65	62.63	49.99
Biba3	93.77	89.32	48.78
Biba4	52.98	51.08	49.09
Biba5	77.00	74.74	49.26
Biba6	113.20	111.4	49.60
BL1	60.78	58.74	49.15
BL2	60.51	60.24	49.89
BL3	91.55	85.62	48.33
BL4	52.98	51.08	49.09
BL5	72.97	68.78	48.52
BL6	109.00	113.20	50.95
Iso1	50.04	48.69	49.32
Iso2	76.55	72.70	48.71
Iso3	335.40	315.40	48.46
Iso4	494.80	485.00	49.50
Low1	7943.20	42670.10	84.31
Low2	8225.80	46991.40	85.10
Low3	26204.20	130252.80	83.25
Low4	7787.50	40478.70	83.87
Low5	14899.20	92938.20	86.18
Low6	27289.60	146022.40	84.25
High1	7693.30	39843.80	83.82
High2	8388.60	48609.30	85.28
High3	25712.60	126464.00	83.10
High4	7755.80	40345.60	83.88
High5	16588.80	105062.40	86.36
High6	27555.80	148070.40	84.31
Chin1	15083.50	86220.80	85.11
Chin2	27258.90	146227.20	84.29
Chin3	51077.10	273510.40	84.26
Chin4	120627.20	829747.20	87.31

Table 6.10: Memory usage comparison of PicoSAT (SAT) and TraceCheck (Check) for reference monitor test functions as listed in Table 6.5.

Test Function	Consumer	Producer	
	max [KiB]	max [KiB]	workload [%]
Biba1	15560.21	39771.23	71.88
Biba2	15707.04	40030.94	71.82
Biba3	15758.95	40329.55	71.90
Biba4	15537.60	39563.36	71.80
Biba5	15740.49	40123.68	71.82
Biba6	15934.00	40832.20	71.93
BL1	15558.98	39771.12	71.88
BL2	15701.57	39989.08	71.81
BL3	15755.04	40167.46	71.83
BL4	15537.59	39563.35	71.80
BL5	15737.11	40283.19	71.91
BL6	15927.10	40824.10	71.94
Iso1	15474.89	39481.84	71.84
Iso2	15585.14	39890.70	71.91
Iso3	16507.90	43490.80	72.49
Iso4	16896.20	45798.60	73.05
Low1	143202.50	159867.49	52.75
Low2	152294.37	168785.67	52.57
Low3	352355.20	353556.40	50.09
Low4	137924.27	154555.50	52.84
Low5	234085.80	245134.40	51.15
Low6	400150.20	400749.40	50.04
High1	139778.28	156693.33	52.85
High2	157482.50	173826.66	52.47
High3	348272.70	349953.00	50.12
High4	137894.05	154687.44	52.87
High5	287622.10	297841.50	50.87
High6	402567.30	402932.20	50.02
Chin1	255986.90	267065.60	51.06
Chin2	393976.40	394418.90	50.03
Chin3	698584.00	679526.10	49.31
Chin4	1996774.40	1928449.40	49.13

Table 6.11: Memory usage comparison of Producer and Consumer for reference monitor test functions as listed in Table 6.5.

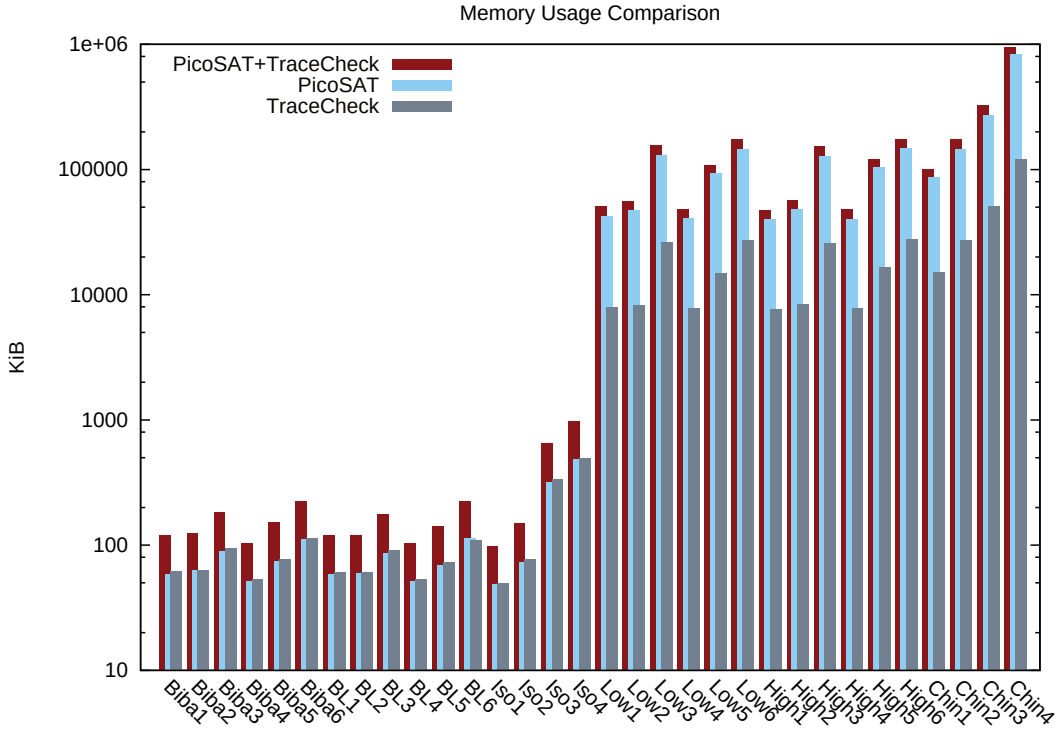


Figure 6.8: Memory usage comparison for verification versus validation. (logarithmic scale)

- For the execution of the complete tool flow, the costs for the consumer are significantly lower for the first (based on static policies) test cases.
- For the execution of the complete tool flow, the consumer's cost are mostly determined by the miter recomputation, which is necessary to assure the match between hardware module, proof, and design specification. A more feasible solution for guaranteeing that match is required to fully embrace the concept of proof-carrying hardware into the prototype tool flow.

The producer has provided 98.86% of the runtime workload for the verification and validation and 76.98% of the runtime for the complete tool flow. I conclude that proof-carrying hardware has the potential to utilize a split between consumer and producer to make formal and other verification feasible to a wide range of consumers that depend on on-the-fly attestation in newly delivered hardware modules. 85.37% of the memory usage for the verification and validation has been shifted to the producer, which shows that other aspects of computational cost can be shifted to the producer as well. The mere 54.66% shift of memory usage for the complete tool flow can be assigned to the miter recomputation task, which already determined the consumer's runtime for the overall tool flow. This indicates that the respective tasks for the consumer and producer have to be

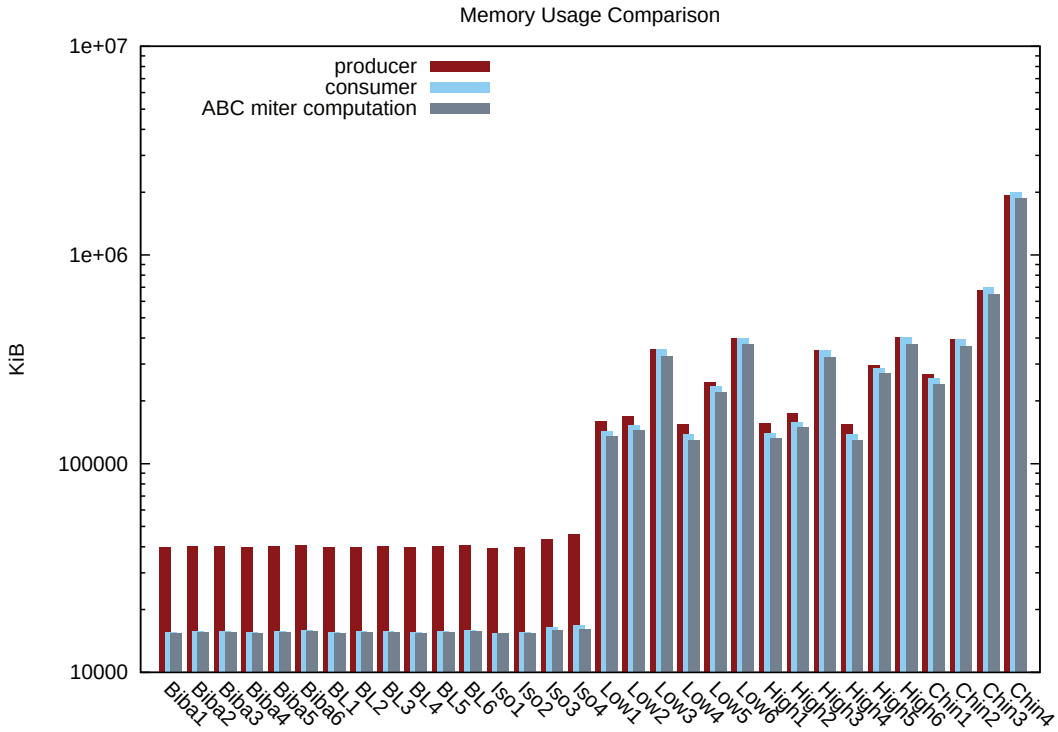


Figure 6.9: Memory usage comparison of producer and consumer workload for test functions listed in Table 6.5. (logarithmic scale)

carefully chosen in order to achieve a scenario where the validation is as light-weight as possible.

Overall, the results are encouraging and show the robustness and feasibility of my novel concept. The next chapter summarizes the contributions of this work and concludes the thesis.

CHAPTER 7

Conclusion and Outlook

7.1 Contributions

With this work, I present the novel concept of proof-carrying hardware, an approach to security for dynamically reconfigurable hardware which utilizes dynamic reconfiguration with hardware modules from untrusted sources. I provide fundamental groundwork to establish trust in such delivered hardware modules through the validation of a safety proof which guarantees the hardware module's absolute adherence to a previously determined safety policy. The benefit of proof-carrying hardware over other approaches to secure the reconfiguration of hardware devices is threefold:

- Reconfigurable hardware scenarios benefit from very short reconfiguration times or even on-the-fly reconfiguration. Yet, they can lack the resources to handle large-scale testing or verification tasks. Even if resources exist and are also available, it may not always be economical to perform elaborate proof computations on site. Those reconfigurable systems benefit from proof-carrying hardware as the producer of the hardware module provides the computationally costly safety proof which the reconfigurable platform quickly validates. This shift of workload away from the consumer to the producer of the hardware module is a principle inherent to the concept of proof-carrying hardware and allows for proof-carrying hardware's great usability.

- Proof-carrying hardware establishes trust in a hardware module delivered by an untrusted producer. The only requirement is a trustworthy procedure at the consumer's disposal to check the proof. The production and verification process is left up to the producer based on the safety policy. In the context of this work, the safety policy is the demand for combinational or sequential equivalence, respectively, and the format of the bitstream. Neither this process nor the transmission of the proof-carrying bitstream require any security measures since the validation of the proof-carrying

bitstream at the consumer side not only checks the safety proof for correctness but also matches the proof, the hardware module, and the safety policy (established by the consumer) against each other. Any manipulation or design flaw resulting in a violation of the safety policy would be noticed by the consumer before he would trust and run the newly delivered hardware. Hence, proof-carrying hardware is robust against malicious tampering or accidental misuse.

- Proof-carrying hardware can be applied to a wide range of security challenges and even combinations of them. The safety policy states formally all the consumer's security needs that the hardware module has to fulfill. The policy incorporates therefore the individual security concerns as well as the individual target platform's specifications. In general, security properties of reconfigurable hardware that can be described and validated, formally or otherwise, are verifiable with proof-carrying hardware due to its great flexibility.

I have provided a flexible and promising concept. I have also provided a proof-carrying hardware prototype as the basis for further extensions of the proof-carrying hardware concept. The prototype also allows for an evaluation of the concept to provide the proof to the thesis claim made in Section 3.5.

- The transfer of the proof-carrying code concept to the domain of reconfigurable hardware has been done with the development of proof-carrying hardware.

- Proof-carrying hardware is usable. The successful shift of workload regarding runtime and memory usage has been demonstrated in Section 6.2 and Section 6.3. Especially for the tasks involved in establishing trust in the security of the new hardware module, the shift of workload has been encouraging.

- I also deliberately and under controlled circumstances manipulated the different files to determine the robustness of the prototype. The results were encouraging as the proof validation and proof matching process was able to detect any manipulation. Proof-carrying hardware's concept shows great robustness against manipulation.

- I have applied proof-carrying hardware to the runtime verification of combinational and bounded sequential equivalence checks and thereby also employed it in the use case scenario of monitoring the reconfigurable system at runtime. This demonstrates to potential flexibility of proof-carrying hardware.

7.2 Conclusions

With the proof-carrying hardware prototype tool flow, I examined the potential of proof-carrying hardware. The prototype applies proof-carrying hardware to the security challenge of verifying combinational and bounded sequential equivalence. The verification of design and implementation equivalence is only a single instance of the proof-carrying hardware concept. My novel approach is a general concept based on the principle that

untrusted agents deliver hardware in combination with a safety proof. This idea is not tied to a certain kind of security challenge or proof. Also, the idea of what constitutes a proof extends beyond formal proofs. In theory, the proof-carrying hardware concept applies also to any kind of security assurance that a consumer of hardware can validate.

The above mentioned prototype tool flow is, as detailed in Chapter 5 based on open-source tools and my abstract FPGA architecture and bitstream format. Proof-carrying hardware would benefit from an additional implementation in propriety tools and FPGA boards. The application to more complex security challenges would extend the concept to adapt to new use cases and demonstrate how well the concept scales to real-life problems.

The aspect of scalability is particularly interesting. Section 6.2 has shown that the complexity of verification can vary greatly for different test cases. Proving combinational equivalence for all n-bit multiplier test functions was actually beyond the capability of the experiment's setup. This is due to the fact that formal verification is usually a problem of high complexity, NP-complete for SAT solving, and can become exponentially difficult. As proof-carrying hardware utilizes existing verification methods and tools, the concept's usability improves with tools that scale well for larger problems.

7.3 Lessons Learned

From the proof-carrying hardware prototype implementation I was able to draw conclusions regarding any implementation of the novel method.

1. For verification and validation of the proof the producer is left with 98.86 % of the runtime workload and 85.37 % of the memory footprint. For the overall workload including every step in the tool flow, the producer delivers 76.98 % of the total runtime and 51.66 % of the memory usage. The conclusion to draw from these results is that
   a) a single task assigned to the consumer, in this case the miter recomputation, can change the shift of workload in favor of the producer, a disadvantage to the consumer which should be avoided.
   b) not all aspects of computational workload shift in the same way: memory usage never falls below a certain minimum once the tool is run and the test file is processed. The runtime measurements show greater variations and do not depend as much on file I/O and are therefore more flexible to shift.
2. Test functions that were more complex involved more runtime and memory usage for both the consumer and producer. Considering only the verification and validation of the resolution proof, the shift of workload to the producer was more successful for the more demanding test cases.

There are two lessons to be learned for the future application of proof-carrying hardware:

1. The actual implementation of proof-carrying hardware influences the usability greatly. Depending on the tools of the tool flow, a single task can determine how successful

the shift of workload is and how resources are used. The specific scenario and the available resources should indicate the choice of tools and methods.

2. The shift of workload varies with the size of the problem. More specifically, the shift of workload begins after a minimum amount of work is performed by the consumer as well as the producer. This minimum amount of work consists of file I/O and starting a program since those tasks have to be executed no matter the size of the actual computational problem. The type of verification to which proof-carrying hardware is applied should be chosen under advisement of the actual size of the verification problem.

7.4 Outlook and Future Work

This thesis has presented proof-carrying hardware as a novel concept for the security of reconfigurable devices and their reconfiguration with bitstreams from untrusted sources. As reconfigurable hardware and its ability to update becomes increasingly important, the security of said updates becomes equally important. Also, the embedded systems and their deployment environments become increasingly complex which leads to an ever-growing complexity of security. From my perspective, proof-carrying hardware is a step towards meeting this new demand for security. This unique and novel approach demonstrates a new understanding of security and security assurance. Considering the many different parties involved in the production of reconfigurable fabric and its hardware modules, it is not manageable to establish trust in a specific producer or supplier. Instead, the goal should be to establish trust in the products delivered, not the producer himself. I am also convinced that proof-carrying hardware's inherent flexibility to extend and its use of open-source tools and formats will make it highly attractive for the research community.

This thesis presented the groundwork for this concept as a first step. As my research has shown encouraging results, future work will further proof-carrying hardware in multiple directions:

- Proof-carrying hardware is particularly suited for the verification of security properties that result in a proof which can be quickly validated. Future work will apply proof-carrying hardware to further scenarios where security is costly to verify but comparably inexpensive to be checked with a safety proof. Security may cover functional properties but particularly non-functional properties of the hardware:
  - Physical isolation, for instance in the form of moats and drawbridges [36] could benefit from proof-carrying hardware. If the consumer of a bitstream containing physically isolated IP cores was given a means of validating the sufficient isolation, trust could be established in the implementation. Other placement and routing constraints could also be verified.
  - A vital aspect of hardware is the clock frequency. It is conceivable to apply proof-carrying hardware to the verification of a minimum, maximum or exact clock frequency to assure the untroubled integration of a hardware module.

- ReconOS [52] is an operating system for dynamically reconfigurable hardware. Proof-carrying hardware has already been applied to the verification of reference monitor modules that manage the access to shared memory. The integration of those verified and validated modules into ReconOS in addition to or exchange for the regular memory management is planned.

- The proof-carrying bitstream itself is an important aspect of this novel concept as it contains both, hardware module and proof. The bitstream's make-up enforces the usability and the robustness of proof-carrying hardware. The bitstream should no longer be made up of two distinct parts. A more feasible solution would be to have the hardware module carry the proof implicitly in its make-up, as suggested by Betz [12].

My novel approach delivers proof-carrying hardware as a flexible concept. This concept is meant for further extension to cover not only a wide range of security challenges but also to deliver to more specific needs. The long-term goal of the development of proof-carrying hardware must be to provide for a means to describe any security threat and physical aspect of reconfigurable hardware security. This shall result in truly customized safety policies and design-specific proofs of safety to cater to a reconfigurable platform's individual security need.

APPENDIX A

File Formats

A.1 Pre-existing Tool Flow File Formats

Listing A.1: Excerpt from blif file for Chines Wall example Listing 4.2.

```
.model State_Machine

.inputs top^clock top^reset top^module_id~0 top^module_id~1 \
 top^module_id~2 top^module_id~3 top^module_id~4 top^op~0 \
top^op~1 top^address~0 top^address~1 top^address~2 \
top^address~3 top^address~4 top^address~5 top^address~6
top^address~7 top^address~8 top^address~9 top^address~10 \
top^address~11 top^address~12 top^address~13 top^address~14 \
top^address~15 top^address~16 top^address~17 top^address~18 \
top^address~19 top^address~20 top^address~21 top^address~22 \
top^address~23 top^address~24 top^address~25 top^address~26 \
top^address~27 top^address~28 top^address~29 top^address~30 \
top^address~31
//global inputs

.outputs top^is_legal
//global outputs

.latch  n151 FF_NODE33  re top^clock 0
.latch  n191 FF_NODE34  re top^clock 0
// latch instances with one input, one output,
// sensitive to the rising edge of the clock
// initiated with the Boolean value 0
```

```
.names FF_NODE33 FF_NODE34 n48
11 1
.names top^address~26 top^address~27 n49
00 1
.names top^address~25 top^address~31 n50
00 1
.names top^address~24 top^address~30 n51
00 1
.names top^address~28 top^address~29 n52
00 1
.names top^address~22 top^address~23 n53
00 1
.names top^address~20 top^address~21 n54
00 1
.names n53 n54 n55
11 1
.names n51 n52 n56
11 1
.names n49 n50 n57
11 1
.names n56 n57 n58
11 1
.names n55 n58 n59
11 1
.names top^address~14 top^address~15 n60
00 1
.names top^address~12 top^address~13 n61
00 1
[...]
// excerpt from the list of logic gate instances
// instances have two inputs and one output
// the logic function is defined by lines from the truth
// table implementing the function, defining either all
// positive or negative outputs is sufficient

.names n48 top^is_legal
0 1
.names n97 n191
0 1
// the number of inputs for the logic gate is not fixed

.end
```

Listing A.2: Excerpt from .net netlist file for Chines Wall Verilog example Listing 4.2.

```
.global top^clock
//globl clock

.input top^clock
pinlist: top^clock

.input top^reset
pinlist: top^reset

.input top^module_id~0    // module ID as global inputs
pinlist: top^module_id~0

.input top^module_id~1
pinlist: top^module_id~1

.input top^module_id~2
pinlist: top^module_id~2

.input top^module_id~3
pinlist: top^module_id~3

.input top^module_id~4
pinlist: top^module_id~4

.input top^op~0           // memory access type as global input
pinlist: top^op~0

.input top^op~1
pinlist: top^op~1

[...]

.input top^address~27     // excerpt of the memory range address
pinlist: top^address~27   // as global inputs

.input top^address~28
pinlist: top^address~28

.input top^address~29
pinlist: top^address~29
```

```
.input top^address~30
pinlist: top^address~30

.input top^address~31     // address length maximum of 32 bits
pinlist: top^address~31


.output out:top^is_legal
pinlist: top^is_legal
// global output

.clb n66
pinlist: n64 n65 open open n66 open
subblock: n66 0 1 open open 4 open

.clb n64
pinlist: top^address~10 top^address~11 open open n64 open
subblock: n64 0 1 open open 4 open

.clb n65
pinlist: top^address~8 top^address~9 open open n65 open
subblock: n65 0 1 open open 4 open

.clb n63
pinlist: n61 n62 open open n63 open
subblock: n63 0 1 open open 4 open

.clb n62
pinlist: n57 n58 open open n62 open
subblock: n62 0 1 open open 4 open

.clb n61
pinlist: n59 n60 open open n61 open
subblock: n61 0 1 open open 4 open

.clb n57
pinlist: top^address~14 top^address~15 open open n57 open
subblock: n57 0 1 open open 4 open

// each configurable logic block (clb) and its subblocks
// are determined with inputs, undefined inputs, and output

[...]
```

Listing A.3: Excerpt from placement file for netlist Listing A.2 resulting from Listing 4.2.

```
 Netlist file: chin1b.net   Architecture file: k4-n1.xml
Array size: 13 x 13 logic blocks

#block name                x       y       subblk   block number
#----------                --      --      ------   ------------
top^clock                  14      13      2        #0
top^reset                  0       3       0        #1
top^module_id~0            0       7       2        #2
top^module_id~1            0       7       0        #3
top^module_id~2            0       11      1        #4
top^module_id~3            0       11      2        #5
top^module_id~4            0       11      0        #6
top^op~0                   0       6       2        #7
top^op~1                   0       6       1        #8
top^address~4              14      3       0        #9
top^address~5              14      2       0        #10
top^address~6              14      2       2        #11
top^address~7              14      2       1        #12
top^address~8              14      5       0        #13
top^address~9              14      5       1        #14
top^address~10             14      7       2        #15
top^address~11             14      7       0        #16
top^address~12             7       14      0        #17
top^address~13             7       14      1        #18
top^address~14             7       14      2        #19
top^address~15             5       14      1        #20
top^address~16             0       9       1        #21
top^address~17             0       9       0        #22
top^address~18             0       10      2        #23
top^address~19             0       10      1        #24
top^address~20             14      8       0        #25
top^address~21             14      8       2        #26
top^address~22             14      9       1        #27
top^address~23             14      9       2        #28
top^address~24             14      12      0        #29
top^address~25             9       14      1        #30
top^address~26             14      11      2        #31
top^address~27             14      11      0        #32
top^address~28             14      11      1        #33
top^address~29             14      10      2        #34
top^address~30             14      12      1        #35
top^address~31             9       14      2        #36
```

out:top^is_legal	11	0	0	#37
n66	13	6	0	#38
n64	13	7	0	#39
n65	13	5	0	#40
n63	11	10	0	#41
n62	7	11	0	#42
n61	2	10	0	#43
n57	6	11	0	#44
n59	1	10	0	#45
n58	8	11	0	#46
n60	1	9	0	#47
n78	10	11	0	#48
n79	10	10	0	#49
n76	11	11	0	#50
n77	9	11	0	#51
n71	13	12	0	#52
n72	12	11	0	#53
n70	9	12	0	#54
n69	13	11	0	#55
top^FF_NODE~88	1	1	0	#56
n194	2	3	0	#57
n195	1	3	0	#58
n196	1	2	0	#59
n193	3	3	0	#60
n68	12	4	0	#61
n67	13	3	0	#62
n190	4	3	0	#63
n95	9	10	0	#64
n94	12	10	0	#65
n106	12	8	0	#66
n93	12	6	0	#67
n105	12	7	0	#68
n100	11	6	0	#69
n92	12	1	0	#70
n90	13	2	0	#71
n91	13	1	0	#72
n99	12	5	0	#73
n104	12	3	0	#74
n98	13	4	0	#75
n103	12	2	0	#76
n96	6	10	0	#77
[...]				
top^is_legal	11	1	0	#182

Listing A.4: Excerpt from routing file for netlist Listing A.2 resulting from Listing 4.2.

```
Array size: 13 x 13 logic blocks.

Routing:

Net 0 (top^clock): global net connecting:

Block top^clock (#0) at (14, 13), Pin class 7.
Block top^FF_NODE~88 (#56) at (1, 1), Pin class 2.
Block top^FF_NODE~85 (#122) at (4, 2), Pin class 2.
Block top^FF_NODE~86 (#145) at (1, 4), Pin class 2.
Block top^FF_NODE~87 (#152) at (3, 2), Pin class 2.

Net 1 (top^reset)
SOURCE (0,3)  Pad: 1
  OPIN (0,3)  Pad: 1
 CHANY (0,2) to (0,3)  Track: 7
  IPIN (1,3)  Pin: 1
  SINK (1,3)  Class: 0
 CHANY (0,2) to (0,3)  Track: 7
 CHANX (1,1) to (2,1)  Track: 6
  IPIN (1,2)  Pin: 0
  SINK (1,2)  Class: 0
  OPIN (0,3)  Pad: 1
 CHANY (0,2) to (0,3)  Track: 3
 CHANX (1,1)  Track: 0
 CHANY (1,2) to (1,3)  Track: 0
 CHANX (2,2) to (3,2)  Track: 2
 CHANY (3,1) to (3,2)  Track: 7
  IPIN (4,2)  Pin: 1
  SINK (4,2)  Class: 0
 CHANX (1,1) to (2,1)  Track: 6
 CHANY (2,2) to (2,3)  Track: 6
 CHANX (3,3) to (4,3)  Track: 6
 CHANY (4,2) to (4,3)  Track: 3
 CHANX (3,2) to (4,2)  Track: 1
  IPIN (3,2)  Pin: 2
  SINK (3,2)  Class: 0
  OPIN (0,3)  Pad: 1
 CHANY (0,3) to (0,4)  Track: 4
  IPIN (1,4)  Pin: 1
  SINK (1,4)  Class: 0
```

```
Net 2 (top^module_id~0)
SOURCE (0,7)  Pad: 7
  OPIN (0,7)  Pad: 7
 CHANY (0,6) to (0,7)  Track: 7
  IPIN (1,7)  Pin: 1
  SINK (1,7)  Class: 0

Net 3 (top^module_id~1)
SOURCE (0,7)  Pad: 1
  OPIN (0,7)  Pad: 1
 CHANY (0,6) to (0,7)  Track: 3
 CHANX (1,5)  Track: 0
 CHANY (1,6) to (1,7)  Track: 0
  IPIN (1,7)  Pin: 3
  SINK (1,7)  Class: 0

[...]

Net 7 (top^op~0)
SOURCE (0,6)  Pad: 7
  OPIN (0,6)  Pad: 7
 CHANY (0,6) to (0,7)  Track: 2
 CHANX (1,6) to (2,6)  Track: 0
  IPIN (1,6)  Pin: 2
  SINK (1,6)  Class: 0

[...]


Net 36 (top^address~31)
SOURCE (9,14)  Pad: 7
  OPIN (9,14)  Pad: 7
 CHANX (9,13) to (10,13)  Track: 2
 CHANY (9,13)  Track: 3
 CHANY (9,11) to (9,12)  Track: 3
  IPIN (9,12)  Pin: 3
  SINK (9,12)  Class: 0

[...]

Net 180 (n158)
SOURCE (4,5)  Class: 1
  OPIN (4,5)  Pin: 4
```

```
 CHANX (3,4) to (4,4)  Track: 5
 CHANY (3,5) to (3,6)  Track: 6
 CHANX (4,6) to (5,6)  Track: 6
  IPIN (4,7)  Pin: 0
  SINK (4,7)  Class: 0

Net 181 (top^is_legal)
SOURCE (11,1)  Class: 1
  OPIN (11,1)  Pin: 4
 CHANX (11,0) to (12,0)  Track: 0
  IPIN (11,0)  Pad: 0
  SINK (11,0)  Pad: 0
```

A.2 Proof-carrying Hardware File Format

Listing A.5: Complete bitstream file for Chines Wall Verilog example Listing 4.2.

```
.begin

block_array_size: 8x8
lut_size: 2

gin:  clock reset module_id0 \
module_id1 module_id2 \
module_id3 module_id4 op0 op1 \
adr0 adr1 adr2 adr3 adr4 adr5 \
adr6 adr7 adr8 adr9 adr10 \
adr11 adr12 adr13 adr14 adr15 \
adr16 adr17 adr18 adr19 adr20 \
adr21 adr22 adr23 adr24 adr25 \
adr26 adr27 adr28 adr29 adr30 \
adr31
// all global inputs

gout: is_legal
// all global outputs

.gout is_legal (7,0).6
// global output routing

.gin clock (3,9)
lout (8,3).default
lout (8,5).default
// special clock routing

.gin reset (9,4)
lout (8,4).3
lout (8,5).3
// regular global input routing

.gin module_id0 (0,3)
lout (1,3).2

.gin module_id1 (0,3)
lout (1,3).1

.gin module_id2 (6,0)
lout (6,1).3

.gin module_id3 (6,0)
lout (6,1).0

.gin module_id4 (5,0)
lout (5,1).3

.gin op0 (0,2)
```

```
lout (1,2).3

.gin op1 (0,2)
lout (1,2).2

.gin adr4 (0,5)
lout (1,5).1
lout (1,4).0

.gin adr5 (0,5)
lout (1,5).2
lout (1,4).2

.gin adr6 (0,6)
lout (1,6).1

.gin adr7 (0,6)
lout (1,6).2

.gin adr8 (5,9)
lout (5,7).1

.gin adr9 (5,9)
lout (5,7).3

.gin adr10 (9,6)
lout (8,6).3

.gin adr11 (9,6)
lout (8,6).1

.gin adr12 (9,6)
lout (7,6).1

.gin adr13 (9,7)
lout (7,6).0

.gin adr14 (9,7)
lout (8,7).3

.gin adr15 (9,7)
lout (8,7).2

.gin adr16 (2,9)
lout (2,7).1

.gin adr17 (2,9)
lout (2,7).3

.gin adr18 (0,7)
lout (1,7).1

.gin adr19 (0,7)
lout (1,7).2

.gin adr20 (3,0)
lout (3,1).3

.gin adr21 (3,0)
lout (3,1).0

.gin adr22 (2,0)
lout (2,2).3

.gin adr23 (0,2)
lout (2,2).1

.gin adr24 (8,0)
lout (8,1).2

.gin adr25 (4,0)
lout (4,1).0

.gin adr26 (0,1)
lout (1,1).3

.gin adr27 (0,1)
lout (1,1).1

.gin adr28 (9,2)
lout (8,2).1

.gin adr29 (9,2)
lout (8,2).3

.gin adr30 (8,0)
```

```
lout (8,1).0

.gin adr31 (4,0)
lout (4,1).3

.latch (8,5)
n151 FF_NODE33
lout (7,3).0
lout (6,2).3
lout (6,3).2
// latch logic block
// names of input and output
// multiple local outputs

.latch (8,3)
n191 FF_NODE34
lout (7,3).2
lout (6,2).2

.lut (7,3)
FF_NODE33 FF_NODE34 n48
11 1
00 1
lout (5,2).1
lout (7,1).3

.lut (1,1)
adr26 adr27 n49
00 1
lout (2,1).1
// lut logic block
// names of inputs and outputs

.lut (4,1)
adr25 adr31 n50
00 1
lout (2,1).2

.lut (8,1)
adr24 adr30 n51
00 1
lout (7,2).1
```

```
.lut (8,2)
adr28 adr29 n52
00 1
lout (7,2).2

.lut (2,2)
adr22 adr23 n53
00 1
lout (3,2).2

.lut (3,1)
adr20 adr21 n54
00 1
lout (3,2).0

.lut (3,2)
n53 n54 n55
11 1
lout (4,3).3

.lut (7,2)
n51 n52 n56
11 1
lout (4,2).0

.lut (2,1)
n49 n50 n57
11 1
lout (4,2).1

.lut (4,2)
n56 n57 n58
11 1
lout (4,3).1

.lut (4,3)
n55 n58 n59
11 1
lout (4,4).3
lout (5,5).2

.lut (8,7)
adr14 adr15 n60
```

```
00 1
lout (6,6).2

.lut (7,6)
adr12 adr13 n61
00 1
lout (6,6).0

.lut (1,7)
adr18 adr19 n62
00 1
lout (2,6).1

.lut (2,7)
adr16 adr17 n63
00 1
lout (2,6).2

.lut (2,6)
n62 n63 n64
11 1
lout (3,6).2

.lut (6,6)
n60 n61 n65
11 1
lout (3,6).0

.lut (3,6)
n64 n65 n66
11 1
lout (4,6).2
lout (4,5).3

.lut (8,6)
adr10 adr11 n67
00 1
lout (5,6).1

.lut (5,7)
adr8 adr9 n68
00 1
lout (5,6).2

.lut (5,6)
n67 n68 n69
11 1
lout (3,4).0
lout (3,5).3

.lut (1,6)
adr6 adr7 n70
00 1
lout (2,4).0
lout (2,5).2

.lut (1,4)
adr4 adr5 n71
01 1
10 1
lout (2,4).2

.lut (2,4)
n70 n71 n72
11 1
lout (3,4).2

.lut (3,4)
n69 n72 n73
11 1
lout (4,5).1

.lut (4,5)
n66 n73 n74
11 1
lout (4,4).2

.lut (4,4)
n59 n74 n75
11 1
lout (5,4).2
lout (7,5).0

.lut (1,3)
module_id0 module_id1 n76
10 1
```

```
lout (2,3).2

.lut (1,2)
op0 op1 n77
10 1
lout (2,3).3

.lut (6,1)
module_id2 module_id3 n78
00 1
lout (5,1).0

.lut (5,1)
module_id4 n78 n79
01 1
lout (3,3).2

.lut (2,3)
n76 n77 n80
11 1
lout (3,3).3

.lut (3,3)
n79 n80 n81
11 1
lout (6,3).3
lout (5,3).0

.lut (1,5)
adr4 adr5 n82
lout (2,5).3

.lut (2,5)
n70 n82 n83
11 1
lout (3,5).2

.lut (3,5)
n69 n83 n84
11 1
lout (4,6).1

.lut (4,6)
n66 n84 n85
11 1
lout (5,5).3

.lut (5,5)
n59 n85 n86
11 1
lout (6,5).2
lout (6,4).1

.lut (6,3)
FF_NODE33 n81 n87
01 1
lout (6,5).0

.lut (6,5)
n86 n87 n88
01 1
lout (7,5).1

.lut (7,5)
n75 n88 n89
11 1
lout (7,4).2
lout (8,5).2

.lut (8,5)
reset n89 n151
00 1

.lut (6,2)
FF_NODE33 FF_NODE34 n91
lout (5,2).0

.lut (5,2)
n48 n91 n92
00 1
lout (5,3).3

.lut (5,3)
n81 n92 n93
11 1
lout (5,4).3
```

```
.lut (5,4)
n75 n93 n94
01 1
lout (6,4).2

.lut (6,4)
n86 n94 n95
11 1
lout (7,4).1

.lut (7,4)
n89 n95 n96
00 1
lout (8,4).2

.lut (8,4)
reset n96 n97
00 1
lout (8,3).2

.lut (7,1)
- n48 is_legal
// first input is not used
-0 1
gout (7,0).6
// global outputs

.lut (8,3)
- n97 n191
-0 1

.end
```

Bibliography

[1] TraceCheck. http://fmv.jku.at/tracecheck/index.html, 2006. [Online; accessed July 2012].

[2] Berkeley Logic Interchange Format (BLIF). http://www1.cs.columbia.edu/~cs4861/s07-sis/blif/index.html, 2007. [Online; accessed July 2012].

[3] SAT Competition 2009: Benchmark Submission Guidelines. http://www.satcompetition.org/2009/format-benchmarks2009.html, 2009. [Online; accessed July 2012].

[4] ABC: A System for Sequential Synthesis and Verification. http://www.eecs.berkeley.edu/~alanmi/abc/, 2011. [Online; accessed April 2012].

[5] The Coq Proof Assistant. http://www.coq.inria.fr, 2011. [Online; accessed December 2011].

[6] U.S. Department of State - Clearances. http://www.state.gov/m/ds/clearances/c10977.htm, 2012. [Online; accessed March 2012].

[7] Wikipedia - List of U.S. security clearance terms. http://en.wikipedia.org/wiki/List_of_U.S._security_clearance_terms, 2012. [Online; accessed March 2012].

[8] B. Badrignans, J.L. Danger, Fischer V., G. Gogniat, and L. Torres, editors. *Security Trends for FPGAS*. Springer, 2011.

[9] B. Badrignans, R. Elbaz, and L. Torres. Secure FPGA Configuration Architecture Preventing System Downgrade. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 317–322. IEEE, September 2008.

[10] D. Beckhoff, C. an Koch and J. Torresen. Short-Circuits on FPGAs caused by Partial Runtime Reconfiguration. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 596–601. IEEE, August / September 2010.

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

[12] V. Betz. Personal Communication, 2010.

[13] V. Betz, T. Campell, W. M. Fang, P. Jamieson, I. Kuon, J. Luu, A. Marquardt, J. Rose, and A. Ye. *VPR and T-VPack User's Manual (Version 5.0)*, July 2009. User's Manual.

[14] V. Betz and J. Rose. VPR: A new Packing, Placement and Routing Tool for FPGA Research. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, volume 1304, pages 213–222. Springer, September 1997.

[15] V. Betz, J. Rose, and A. Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999.

[16] A Biere. *TraceCheck Manual*. Johannes Kepler University, Linz, Austria, 2006. [Online; accessed July 2012].

[17] A. Biere. PicoSAT Essentials. In *Journal on Satisfiability , Boolean Modeling and Computation*, volume 4, pages 75–97. Delft University, 2008.

[18] Armin Biere and Carla Gomes, editors. *Extended Resolution Proofs for Symbolic SAT Solving with Quantification*, volume 4121 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2006.

[19] R. Brayton and A. Mishchenko. ABC: An Academic Industrial-Strength Verification Tool. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 24–40. Springer Berlin / Heidelberg, 2010.

[20] S. Chatterjee, A. Mishchenko, R. Brayton, and A. Kuehlmann. On Resolution Proofs for Combinatorial Equivalence. In *Proceedings of the Annual Design Automation Conference (DAC)*, pages 600–605. ACM/IEEE, June 2007.

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

[22] C. Colby, P. Lee, G.C. Necula, and M. Plesko. A Proof-Carrying Code Architecture for Java. In *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 557–560. Springer Berlin / Heidelberg, 2000.

[23] Altera Corporation. Introduction to the Quartus II Software Version 10.0. Manual, 2010. MNL-01055-1.0.

[24] F. Devic, L. Torres, and B. Badrignans. Secure Protocol Implementation for Remote Bitstream Update Preventing Replay Attacks on FPGA. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 179–182. IEEE, August / September 2010.

[25] S. Drimer. Authentication of FPGA Bitstreams: Why and How. In P. Diniz, E. Marques, K. Bertels, M. Fernandes, and J. Cardoso, editors, *Reconfigurable Computing: Architectures, Tools and Applications*, volume 4419 of *Lecture Notes in Computer Science*, pages 73–84. Springer Berlin / Heidelberg, 2007.

[26] S. Drimer and M.G. Kuhn. A Protocol for Secure Remote Updates of FPGA Configurations. In Jürgen Becker, Roger Woods, Peter Athanas, and Fearghal Morgan, editors, *Reconfigurable Computing: Architectures, Tools and Applications*, volume 5453 of *Lecture Notes in Computer Science*, pages 50–61. Springer Berlin / Heidelberg, 2009.

[27] S. Drzevitzky. Proof-Carrying Hardware: Runtime Formal Verification for Secure Dynamic Reconfiguration. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 255–258. IEEE, August / September 2010. PhD Forum Presentation.

[28] S. Drzevitzky, U. Kastens, and M. Platzner. Proof-Carrying Hardware for Online Verification. *International Journal of Reconfigurable Computing (IJRC)*, 2010:11, 2010.

[29] S. Drzevitzky and Platzner M. Achieving Hardware Security for Reconfigurable Systems on Chip by a Proof-Carrying Code Approach. In *Proceedings of the International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*. IEEE, June 2011.

[30] S. Drzevitzky, M. Platzner, and U. Kastens. Proof-carrying Hardware: Towards Runtime Verification of Reconfigurable Modules. In *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 189–194. IEEE, December 2009.

[31] N. Een, A. Mishchenko, and N. Sörensson. Applying Logic Synthesis to Speedup SAT. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 4501, pages 272–286, 2007.

[32] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *Journal of the ACM (JACM)*, 40(1):143–184, January 1993.

[33] J. Hennessy and D. Patterson. *Computer Organizatoin & Design*. Morgan Kaufmann, second edition, 1998.

[34] 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 (TODAES)*, 13(3):1–24, July 2008.

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

[36] 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 *Proceedings of the Symposium on Security and Privacy (SP)*, pages 281–295. IEEE, May 2007.

[37] T. Huffmire, C. Irvine, T.D. Nguyen, T. Levin, R. Kastner, and T. Sherwood. *Handbook of FPGA Design Security*. Springer, 1st edition, 2010.

[38] 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)*, 3(10):1–35, May 2010.

[39] T. Huffmire, S. Prasad, T. Sherwood, and R. Kastner. Policy-Driven Memory Protection for Reconfigurable Hardware. In *Proceedings of the European Symposium on Research in Computer Security*, volume 4189 of *LNCS*, pages 461–478. Springer, September 2006.

[40] T. Huffmire, T. Sherwood, R. Kastner, and T. Levin. Enforcing Memory Policy Specifications in Reconfigurable Hardware. *Computers & Security*, 27(5-6):197–215, October 2008.

[41] Xilinx Inc. Virtex-4 FPGA User Guide. User Guide, December 2008.

[42] Xilinx Inc. XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices. User Guide, October 2011.

[43] C.E. Irvine and K. Levitt. Trusted Hardware: Can It Be Trustworthy? In *Proceedings of the Annual Design Automation Conference (DAC)*, pages 1–4. ACM, ACM/IEEE, June 2007.

[44] R. Karri, J. Rajendran, K. Rosenfeld, and M. Tehranipoor. Trustworthy Hardware: Identifying and Classifying Hardware Trojans. *Computer*, 43(10):39–46, October 2010.

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

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

[47] K. Klohs. *Validation of Data Flow Results for Program Modules*. PhD thesis, Universität Paderborn, 2009.

[48] K. Klohs and U. Kastens. Memory Requirements of Java Bytecode Verification on Limited Devices. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 132(1):95–111, May 2005.

[49] B. Korte and J. Vygen. *Combinatorial Optimization*, volume 21 of *Algorithms and Combinatorics*. Springer Berlin / Heidelberg, 4 edition, 2008.

[50] E. Love, Y. Jin, and Y. Makris. Proof-Carrying Hardware Intellectual Property: A Pathway to Trusted Module Acquisition. *IEEE Transactions on Information Forensics and Security*, 7(1):25–40, February 2012.

[51] E. Love, Yier Jin, and Y. Makris. Enhancing Security via Provably Trustworthy Hardware Intellectual Property. In *Proceedings of the International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 12–17. IEEE, June 2011.

[52] E. Lübbers and M. Platzner. ReconOS: An Operating System for Dynamically Reconfigurable Hardware. In M. Platzner, J. Teich, and N. Wehn, editors, *Dynamically Reconfigurable Systems*, pages 269–290. Springer Netherlands, 2010.

[53] A. Mishchenko and R. K. Brayton. Recording Synthesis History for Sequential Verification. In *Proceedings of the Confererence on Formal Methods in Computer-Aided Design (FMCAD)*, pages 1–8. IEEE, November 2008.

[54] A. Mishchenko, S. Chatterjee, and R. Brayton. DAG-Aware AIG Rewriting: A Fresh Look at Combinational Logic Synthesis. In *Proceedings of the Annual Design Automation Conference (DAC)*, pages 532–535. ACM, July 2006.

[55] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een. Improvements to Combinational Equivalence Checking. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 836–843. ACM, IEEE/ACM, November 2006.

[56] G. Necula. A Scalable Architecture for Proof-Carrying Code. In Herbert Kuchen and Kazunori Ueda, editors, *Functional and Logic Programming*, volume 2024 of *Lecture Notes in Computer Science*, pages 21–39. Springer Berlin / Heidelberg, 2001.

[57] G.C. Necula and P. Lee. Proof-Carrying Code. Technical Report CMU-CS-96-165, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, November 1996.

[58] J. Peter, B. Kenneth, G. Farnaz, and S. Lesley. Odin II - An Open-Source Verilog HDL Synthesis Tool for CAD Research. In *Proceedings of the Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 149–156. IEEE, April 2010.

[59] H. Savoj, D. Berthelot, A. Mishchenko, and R. Brayton. Combinational Techniques for Sequential Equivalence Checking. In *Proceedings of the Conferenc on Formal Methods in Computer-Aided Design (FMCAD)*, pages 145–149. IEEE, October 2010.

[60] R. Schneck and G. Necula. A Gradual Approach to a More Trustworthy, yet Scalable, Proof-Carrying Code. In *Proceedings of the Conference on Automated Deduction*, volume 2392 of *LNCS*, pages 47–62. Springer Berlin / Heidelberg, July 2002.

[61] J. Seward, N. Nethercote, and T. Hughes. *Valgrind Documentation*, August 2009.

[62] S. Singh and C.J. Lillieroth. Formal Verification of Reconfigurable Cores. In *Proceedings of the Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 25–32. IEEE, April 1999.

[63] M. Tehranipoor and C. Wang, editors. *Introduction to Hardware Security and Trust*. Springer Verlag, 2012.

[64] T. Todman and W. Luk. Verification of Streaming Designs by Combining Symbolic Simulation and Equivalence Checking. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 234–249. IEEE, August 2012.

[65] N. Whitehead, M. Abadi, and G. Necula. By Reason and Authority: a System for Authorization of Proof-Carrying Code. In *Proceedings of the Computer Security Foundations Workshop*, pages 236–250. IEEE, June 2004.

[66] Z. Yu, S. Zeng, Y. Guo, N. Hu, and L. Song. Pathfinder Based on Simulated Annealing for Solving Placement and Routing Problem. In *Proceedings of the International Conference on Advances in Computation and Intelligence*, volume 6382 of *LNCS*, pages 390–401. Springer Berlin / Heidelberg, October 2010.