Ted Huffmire
Cynthia Irvine
Thuy D. Nguyen
Timothy Levin
Ryan Kastner
Timothy Sherwood

Handbook of FPGA Design Security

Springer

Handbook of FPGA Design Security

Ted Huffmire • Cynthia Irvine • Thuy D. Nguyen •
Timothy Levin • Ryan Kastner • Timothy Sherwood

Handbook of FPGA Design Security

Springer

Dr. Ted Huffmire
Department of Computer Science
Naval Postgraduate School
Cunningham Road 1411
93943 Monterey, CA
USA
tdhuffmi@nps.edu

Dr. Cynthia Irvine
Department of Computer Science
Naval Postgraduate School
Cunningham Road 1411
93943 Monterey, CA
USA

Thuy D. Nguyen
Department of Computer Science
Naval Postgraduate School
Cunningham Road 1411
93943 Monterey, CA
USA

Timothy Levin
Department of Computer Science
Naval Postgraduate School
Cunningham Road 1411
93943 Monterey, CA
USA

Dr. Ryan Kastner
Dept. of Computer Science and Eng.
University of California, San Diego
Gilman Drive 9500
92093 La Jolla, CA
USA
kastner@cs.ucsd.edu

Dr. Timothy Sherwood
Department of Computer Science
UC, Santa Barbara
93106 Santa Barbara
USA

ISBN 978-90-481-9156-7 e-ISBN 978-90-481-9157-4
DOI 10.1007/978-90-481-9157-4
Springer Dordrecht Heidelberg London New York

Library of Congress Control Number: 2010930170

Cover design: eStudio Calamar

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

To our teachers

Preface

The purpose of this book is to provide a practical approach to managing security in FPGA designs for researchers and practitioners in the electronic design automation (EDA) and FPGA communities, including corporations, industrial and government research labs, and academics. This book combines theoretical underpinnings with a practical design approach and worked examples for combating real world threats. To address the spectrum of lifecycle and operational threats against FPGA systems, a holistic view of FPGA security is presented, from formal top level specification to low level policy enforcement mechanisms, which integrates recent advances in the fields of computer security theory, languages, compilers, and hardware. The net effect is a diverse set of static and runtime techniques that, working in cooperation, facilitate the composition of robust, dependable, and trustworthy systems using commodity components.

We wish to acknowledge the many people who helped us ensure the success of our work on reconfigurable hardware security. In particular, we wish to thank Andrei Paun and Jason Smith of Louisiana Tech University for providing us with a Linux-compatible version of Grail+. We also wish to thank those who gave us comments on drafts of this book, including Marco Platzner of the University of Paderborn, and Ali Irturk and Jason Oberg of the University of California, San Diego. This research was funded in part by National Science Foundation Grant CNS-0524771 and NSF Career Grant CCF-0448654.

Monterey, CA, USA Ted Huffmire
Cynthia Irvine
Thuy D. Nguyen
Timothy Levin
La Jolla, CA, USA Ryan Kastner
Santa Barbara, CA, USA Timothy Sherwood

Ted Huffmire is an assistant professor of computer science at the Naval Postgraduate School in Monterey, California. His research spans both computer security and computer architecture, focusing on hardware-oriented security and the development of policy enforcement mechanisms for application-specific devices. He has a Ph.D. in computer science from the University of California, Santa Barbara. He is a member of the IEEE and the ACM.

Cynthia Irvine is the director of the Center for Information Systems Security Studies and Research (CISR) and a professor of computer science at the Naval Postgraduate School in Monterey, California. Her research interests include high-assurance security. She has a Ph.D. in astronomy from Case Western Reserve University. She is a member of the IEEE, the ACM, and the Astronomical Society of the Pacific.

Thuy D. Nguyen is a senior researcher of computer science at the Naval Postgraduate School in Monterey, California. Her research interests include high-assurance platforms, trusted operating systems, dynamic security services, multilevel security, security evaluation, and security requirements engineering. She has a B.A. in computer science from the University of California, San Diego.

Timothy Levin is an associate research professor at the Naval Postgraduate School in Monterey, California. His research interests include design, analysis and verification of high-assurance security architectures and dynamic security policies. He has a B.S. in computer science from the University of California, Santa Cruz. He is a member of the IEEE and the ACM.

Ryan Kastner is an associate professor in the Department of Computer Science and Engineering at the University of California, San Diego. His research interests focus on many aspects of embedded computing systems, including reconfigurable architectures, digital-signal processing, and security. He has a Ph.D. in computer science from the University of California, Los Angeles.

Timothy Sherwood is an associate professor in the Department of Computer Science at the University of California, Santa Barbara. His research interests include computer architecture, specifically in the development of novel high-throughput methods by which systems can be constructed, monitored, and analyzed. He has a Ph.D. in computer science and engineering from the University of California, San Diego. He is a member of the IEEE and the ACM.

Contents

Acronyms

ACL	Access Control List.
ACM	Association for Computing Machinery.
AES	Advanced Encryption Standard. A common symmetric crypto algorithm.
API	Application Programming Interface.
ASIC	Application-Specific Integrated Circuit. An integrated circuit with “hard-wired” functionality intended for use in one or a limited number of applications.
BBV	Basic Block Vector.
B&L	Bell-LaPadula confidentiality model.
BRAM	Block Random Access Memory.
C&A	Certification and Accreditation.
CAD	Computer-Aided Design.
CAP	Capability Protection.
CC	Common Criteria.
CCEVS	Common Criteria Evaluation and Validation Scheme.
CIA	Confidentiality-Integrity-Availability triad.
CLB	Configuration Logic Block. The basic repeated building block of an FPGA, usually consisting of LUTs, registers, and other logic.
CM	Configuration Management.
CMP	Chip Multiprocessor.
COI	Conflict-of-Interest Class.
COTS	Commercial Off-the-Shelf.
CPU	Central Processing Unit. A general-purpose processor.
DARPA	Defense Advanced Research Projects Agency.
DFA	Deterministic Finite Automaton.
DoS	Denial-of-Service.
DRAM	Dynamic Random Access Memory.
DSP	Digital Signal Processing.
DVI	Digital Visual Interface.

EAL	Evaluation Assurance Level.
EDK	Embedded Development Kit.
EPROM	Erasable Programmable Read-Only Memory.
EEPROM	Electrically Erasable Programmable Read-Only Memory.
ESL	Electronic System Level. ESL Design involves the compilation of a high-level specification of a system to a low-level hardware implementation.
FEMA	Federal Emergency Management Agency.
FFT	Fast Fourier Transform.
FIPS	Federal Information Processing Standard.
FPGA	Field Programmable Gate Array. A reconfigurable hardware device created as an array of logic blocks tied together with a programmable interconnect.
FSA	Finite State Automaton.
FTLS	Formal Top Level Specification. A description of a policy that specifies the legal sharing of memory among cores on an FPGA.
HDL	Hardware Description Language. A language for designing hardware components.
I&A	Identification and Authentication.
IBM	International Business Machines.
IC	Integrated Circuit.
ICAP	Internal Configuration Access Port. An interface to an FPGA for dynamic partial reconfiguration.
IDS	Intrusion Detection System.
IEEE	Institute of Electrical and Electronics Engineers.
I/O	Input/Output.
IOB	Input/Output Block.
IP	Intellectual Property. Hardware modules that are the building blocks of an embedded design.
ISA	Instruction Set Architecture.
ISE	Integrated Synthesis Environment.
IT	Information Technology.
LCD	Liquid Crystal Display.
LED	Light-Emitting Diode.
LFU	Least Frequently Used.
LPSK	Least Privilege Separation Kernel.
LRU	Least Recently Used.
LUT	Look-Up Table. The smallest FPGA logic component, the LUT can be programmed to imitate any logic gate by directly storing the truth table for that gate.
LVS	Layout-versus-schematic. Comparison tools recommended by Trimberger for detecting subversion in FPGA designs via non-destructive validation of the equivalence between the original design and the implemented design.

MATLAB	Matrix Laboratory. Mathematical software that can be used for developing DSP algorithms.
MIMO	Multiple Input Multiple Output.
MLS	Multilevel Security. The science of building systems that can process data with different security labels (e.g., SECRET and UNCLASSIFIED).
NFA	Nondeterministic Finite Automaton.
NIAP	National Information Assurance Partnership.
NIST	National Institute of Standards and Technology.
NP	Nondeterministic Polynomial.
NRE	Non-Recurring Engineering. The masks used in ASIC fabrication are a large part of the NRE cost of ASICs.
NSA	National Security Agency.
NSTISSP	National Security Telecommunications and Information Systems Security Policy.
OPB	On-chip Peripheral Bus.
PCI	Peripheral Component Interconnect.
PKI	Public Key Infrastructure.
PUF	Physical Unclonable Function. A unique number generated from variations in the manufacturing process.
RAM	Random Access Memory.
RISC	Reduced Instruction Set Computer.
RVM	Reference Validation Mechanism.
RSA	Rivest Shamir Adelman. A common asymmetric key crypto algorithm.
SDK	Software Development Kit.
SGI	Silicon Graphics, Inc.
SKPP	Separation Kernel Protection Profile.
SMP	Symmetric Multiprocessor.
SMT	Simultaneous Multithreading.
SoC	System-on-a-Chip.
SRAM	Static Random Access Memory.
SRC	Seymour Roger Cray. Founder of SRC Computers.
SSL	Secure Sockets Layer.
TCB	Trusted Computing Base.
TCSEC	Trusted Computer System Evaluation Criteria.
TIC	TRUST in Integrated Circuits. A DARPA program concerned with hardware subversion.
TLB	Translation Lookaside Buffer.
TOE	Target of Evaluation.
TSF	TOE Security Functionality. A set consisting of all hardware, software, and firmware of the TOE that must be relied upon for the correct enforcement of the security functional requirements.
USB	Universal Serial Bus.
VHDL	VHSIC (Very-High-Speed Integrated Circuits) Hardware Description Language.
VLIW	Very Long Instruction Word.

VM	Virtual Machine.
VMM	Virtual Machine Monitor.
VPN	Virtual Private Network.
VPR	Versatile Place and Route.
WAP	Wireless Access Point.
XPS	Xilinx Platform Studio.

Chapter 1
Introduction and Motivation

Abstract From Bluetooth transceivers to the NASA Mars Rover, FPGAs have become one of the mainstays of embedded system design. By merging properties of hardware and software, reconfigurable devices provide an attractive tradeoff between the performance of application-specific hardware and the programmability of CPUs. Although this flexibility allows developers to quickly prototype and deploy embedded systems with performance close to ASICs, this programmability can also be exploited to disrupt critical functionality, eavesdrop on encrypted communication, or even destroy a chip. Creating systems which are both efficient and flexible, yet fundamentally sound from a security point of view, is an exceedingly challenging endeavor for both researchers and practitioners. All too often the security aspects of a reconfigurable design are not addressed until far too late in the design process, resulting in systems that are protected only by their obscurity. This chapter presents an overview of Field Programmable Gate Array (FPGA) technologies from the viewpoint of security, specifically how and why these devices have grown in importance over the last decade to become one of the most trusted and critical elements of modern computer systems. This chapter also discusses their changing role from a platform for prototyping to a deployable solution, the architecture of a modern FPGA, the security ramifications of their increased use, and some of the lessons from the security community that may be applicable in this domain.

1.1 The Growing Reliance on FPGAs

FPGAs are at the heart of many mission critical devices, silently controlling everything from wireless access points (WAP) to commercial face recognition systems. Unlike the sequential execution provided by a general purpose processor, modern Field Programmable Gate Arrays can perform hundreds of multiplies and thousands of adds each cycle, giving them the computational power to host many different logic modules at the same time. For example, an FPGA-hosted Wireless Access Point (WAP) may employ a signal processing core, a protocol pro-

T. Huffmire et al., *Handbook of FPGA Design Security*,
DOI 10.1007/978-90-481-9157-4_1, © Springer Science+Business Media B.V. 2010

cessing engine, and a packet scheduler, all sharing the same physical silicon. Furthermore, because reconfigurable hardware can be rewritten in both the lab and in the field, rapid design cycles are possible, and patches can even be downloaded to devices already deployed (e.g. bug fixes or functionality enhancements can be pushed out over the network to cell phones or wireless access points on demand).

Because of this rare combination of computation power and flexibility, reconfigurable devices are now the workhorses behind a broad variety of performance-critical embedded systems [9, 15, 19, 40, 51, 62]. In fact, many reconfigurable machines achieve 100x speedups and 100x performance gain per unit of area as compared to a similar microprocessor [12, 18, 75]. Satellites, set-top boxes, intrusion detection systems, the electrical power grid, cryptography units, aircraft, and even the Mars Rover all rely on Field Programmable Gate Arrays (FPGAs) to perform their respective functions. It is estimated that in 2005 alone there were over 80,000 different commercial FPGA design projects started [53]. The bit-level reconfigurability of these devices can be used to implement highly optimized circuits for everything from encryption to FFTs, or even entire customized multi-processor systems. In this section we describe a couple of these different domains and how FPGAs are used in them.

Design Tip: *Benefits of FPGAs*. FPGAs are ideal for rapid protyping of embedded designs and the development of novel computer architectures. The increasing cost of ASIC manufacturing, the performance advantages of FPGAs over general-purpose processors, and the narrowing performance gap between FPGAs and ASICs have resulted in the growing use of FPGAs in real systems. For low-volume segments of the market, such as highly trustworthy systems, FPGAs provide both cost and security advantages over ASICs. For example, with FPGAs, sensitive designs are never sent to a foundry where they could be stolen. The parallelism available on FPGAs also makes them an attractive alternative to general-purpose CPUs for throughput-driven applications.

1.1.1 FPGAs for Aerospace

Because FPGAs are able to provide a useful balance between performance, cost, and flexibility, many avionics systems now make use of them. For example, FPGAs perform crucial functions in the Joint Strike Fighter [56], the new Boeing 787 Dreamliner [20], and the NASA Mars Rover [23, 57]. In these applications, FPGAs are used for cockpit displays, flight management, avionics, weapons guidance, and flight radar [2].

Design Tip: *FPGA Basics*. A *bitstream* specifies how to set the configuration bits of the FPGA fabric. A *core* is a circuit that is used as a building block of a larger embedded system. An *antifuse-based* FPGA uses fuses as the configuration bits; therefore, it can only be programmed once and is nonvolatile thereafter. An *SRAM-based* FPGA uses volatile SRAM cells as the configuration bits. A *flash-based* FPGA uses EEPROM cells as the configuration bits.

The circuits may be either antifuse-based, flash-based, or SRAM-based. While fuse-based circuits are write-once devices, SRAM- and flash-based FPGAs can be written many times, either in the lab or in the field. Flash-based circuits provide low power advantages.

Design Tip: *Choosing an FPGA Type*. SRAM-based, flash-based, and antifuse-based FPGAs have different security properties [76]. Despite the limitation that it is a write-once technology, an antifuse FPGA offers the advantage that theft of the design requires a painstaking and destructive *sand-and-scan* attack, involving removal of the packaging and the progressive etching and electron micrography of each layer to create a 3-D image of the chip. Since the fuses are nonvolatile, the bitstream does not have to be loaded from off-chip, exposing it to board-level probing attacks and attacks on the bitstream encryption mechanisms. Flash-based FPGAs also can store the bitstream on-chip, and the design does not have to be loaded from off-chip. This eliminates one avenue for attackers. However, unlike an antifuse-based FPGA, the design is changeable since flash memory can be modified. In addition, flash-based FPGAs are much easier to probe, and probing attacks are much cheaper to carry out than sand-and-scan attacks. SRAM-based FPGAs must load the bitstream every time they are powered on, and soft memory errors [28] or flaws in the implementation of bitstream decryption mechanisms may provide an opportunity for a well-funded adversary to extract the design. When powered continuously, SRAM-based FPGAs are similar to non-volatile FPGAs in that the design does not have to be loaded from non-volatile off-chip memory.

Consider the example of military avionics, in which a single chip processes both classified targeting information and unclassified fueling and maintenance information. Other multilevel security (MLS) scenarios for avionics include the sensor-shooter problem, in which the intelligence analysts who decide on targets have higher clearances than the soldiers ordered to attack those targets. In another MLS scenario, a coalition member flies in formation with his or her allies, and a policy must specify what information may be shared with each of them. In these multilevel systems, a CPU can be *allocated* to handle a particular level (or a range of levels) of data, and it is assigned a security label (or a security label range). A separate device for each security level adds too much weight to an aircraft. To minimize weight, a single device that can process data at multiple levels is attractive, but without careful attention to security, can be dangerous. Keeping the different levels of information separate requires careful design. Since reconfigurable systems often lack the memory protection, virtual memory, and other traditional separation mechanisms available in a general-purpose system, security techniques are needed to prevent classified data from mixing with unclassified data. In addition, like software update mechanisms, it is critical that the process for remotely updating these devices is very secure to prevent sabotage. Finally, due to the sensitive nature of the intellectual property, it is very important to keep competitors or enemies from being able to reverse engineer these systems easily.

1.1.2 FPGAs for Supercomputing

While desktop computers continue to make incredible gains in performance, there are always problems that lie outside the capabilities of these machines, and scientists and engineers eventually turn to “big iron” when performance is needed. Many supercomputer companies, including SRC Computers [25, 71], Cray [58], and SGI [67, 68], have integrated reconfigurable hardware into their systems to improve performance [10, 16]. A good example of such a system is Cray’s XD1 architecture, which combines six large Xilinx FPGAs (Virtex-4) with twelve x86 processors in each chassis. When an application is loaded on the machine, it includes a bitstream for programming the associated reconfigurable hardware. Although the past generations of FPGAs were not cost competitive with microprocessors in delivering double precision floating point [74], they can provide significant improvements (100x) in integer dominated applications. The current generation of FPGAs includes more integrated support for floating point. In the supercomputing environment, often the code and data being run are either sensitive intellectual property or even classified in nature, requiring a commensurately secure computing environment. Furthermore, supercomputing centers require strong physical security because they are very high profile targets for intruders.

The SRC Reconfigurable computer is an example of a system that uses FPGAs to provide acceleration for programs running on general-purpose processors [25, 71]. Logging into the machine is via a traditional Unix shell interface. A project folder contains both Verilog and either C or Fortran code. A Makefile invokes a Verilog compiler for the Verilog code (resulting in a bitstream), and it also invokes a C compiler for the C code (or a Fortran compiler for the Fortran code). Executing a program on the SRC requires loading the bitstream onto the reconfigurable hardware and loading the executable program onto the general-purpose hardware. From a security standpoint, if the host OS, application software, or user account has been compromised, a malicious bitstream could be loaded onto the reconfigurable hardware. The malicious hardware could interfere with the correct function of the application or even damage the hardware. Since FPGAs are part of a larger system, the security analysis needs to consider the interactions of the CPUs and FPGAs.

1.1.3 FPGAs for Video Analysis

FPGAs also provide a natural fit for complex high speed signal processing applications such as video analysis and face recognition [59]. These algorithms are most often dominated by large matrix operations and are throughput-driven, meaning that parallelism and pipelining are likely to yield large performance gains for these applications. In terms of the security ramifications of these systems, consider the problem of video redaction.

Redaction involves removing sensitive information from data such as documents, songs, and movies. Redaction may be used in the process of making unclassified portions of a secret document available to the public, or may be used to protect people's privacy. An example of the redaction of video involves blurring the faces of people captured by surveillance cameras. This is necessary when the system is being tested or maintained by a person who lacks the necessary authorization to view faces. IBM has developed such a video privacy system called PeopleVision [65]. To implement such a system on an FPGA, at least three IP cores are used: a video core for processing the video, a redaction core for blurring the faces, and an Ethernet core for transmitting the redacted video to the security guard's terminal. Each core requires off-chip memory, and the privacy of data stored in off-chip memory must be protected. For example, the video core must not be able to bypass the redaction core and send data directly to the Ethernet core. Since few applications are developed fully *in-house*, the trustworthiness of the building blocks of a composed embedded system is a growing concern, particularly since reuse of intellectual property is common in both software and hardware development.

1.1.4 FPGAs for High-Throughput Cryptography

Implementing crypto on FPGAs offers several advantages. Block ciphers require many bit-level operations, such as shifting or permuting bits, which can be efficiently implemented on an FPGA. FPGAs also allow algorithm parameters to be changed easily, or the entire circuit can be replaced completely. For example, if mathematicians discover a flaw in a cipher, the bitstream can easily be updated with a patched version. These advantages have been exploited in FPGA implementations of MD5 [17], SHA-2 [70], and a range of other crypto functions [17, 37, 43, 55, 60, 64, 66, 70, 77]. FPGAs are also useful for public-key crypto, such as RSA, in which the fundamental operation is modular multiplication, and Elliptic Curve Crypto, in which the fundamental operation is point multiplication [27, 29, 48, 54]. Reconfigurable devices are also widely used in network intrusion detection systems (IDS) because of the ability to search packet streams at high throughput against multiple rule sets in parallel [4–6, 13, 14, 21, 26, 36, 72].

Most modern symmetric key cryptosystems can be characterized as performing a repeating series of rounds over a given input. Consider the *rotate* operation as an example. A rotate operation shifts the b bits of a word over n positions. The bit that was at position i before the rotate will now be at position $(i + n) \bmod b$. Implementing this in software requires multiple instructions to shift and wrap the bits in the word. In contrast, an FPGA can implement this by simply rearranging the wires so that the bits arrive in their new order. All of this work focuses on using FPGAs to achieve high speed crypto but does not address the security of the reconfigurable systems themselves. Since crypto devices naturally handle secrets (e.g., keys), they are attractive targets for adversaries. A poorly implemented system will be vulnerable to a variety of timing and side channel attacks, allowing attackers to obtain secret keys. Ensuring the integrity of systems is also essential to prevent attackers from modifying the crypto functions to their advantage.

1.1.5 FPGAs for Intrusion Detection and Prevention

Another area related to security where reconfigurable devices are widely employed are network intrusion detection systems (IDS). Because many IDSs require that every byte of every packet be scanned for known attacks or suspicious behavior, intrusion detection is a computationally difficult problem. Network speeds now operate in the realm of gigabits per second, and an intrusion detection scheme must be able to *always* keep up with the network load. This worst-case performance requirement, coupled with the pipelined manner with which analysis is performed, makes FPGAs a nearly perfect design choice. In fact, a wide variety of FPGA-based IDSs have been built [4–6, 13, 14, 21, 26, 36, 72]. It is important that the integrity of the IDS is maintained and that the IDS cannot be bypassed (for example by routing communication around the IDS core).

1.2 FPGA Architectures

On a continuum between general-purpose processors and application-specific integrated circuits (ASICs), FPGAs lie somewhere in the middle. A CPU is a jack-of-all-trades but a master of none: it can run arbitrary code, but this generality comes at a large performance cost. While compilers and programming languages have transformed computer science, making it possible for humans to easily program computers, we often forget about the high overhead of generality. ASICs, on the other hand, can achieve impressive throughput by harnessing parallelism in an optimized circuit, but they are enormously expensive to fabricate (the cost increases every year), requiring large capital investments. An ASIC is a master of one trade: all functions are hard-wired. Reconfigurable hardware, unlike ASICs and CPUs, offers the opportunity to implement a custom circuit without the expense of fabricating silicon, providing improvements in throughput on the order of one hundred times when compared with a CPU [12, 18, 75]. This section describes the nuts and bolts of a typical reconfigurable chip to illustrate its performance advantages and to provide the background needed to understand the security mechanisms for FPGAs discussed in this book.

1.2.1 The Attractiveness of Reconfigurable Hardware

The roots of reconfigurable systems can be traced to Gerald Estrin's work at UCLA in the 1960s. Estrin's "fixed plus variable structure computer" [24] consisted of a standard processor augmented by an array of reconfigurable hardware. His idea was well ahead of the technology at that time, so he was only able to make a crude approximation of his vision.

Interest in reconfigurable systems was renewed in the mid-eighties with the emergence of programmable logic devices, which were used almost exclusively as a fast prototyping device for application specific integrated circuit (ASIC) designers. They allowed the designer to *compile* the application to reconfigurable hardware to determine whether the application exhibited the correct functionality. The prototyping removed the costly step of fabricating the circuit, especially when fabrication yielded a device that exhibited incorrect functionality (i.e. buggy hardware). Additionally, the rapid prototyping lessened the need for intense simulation to verify correctness. If the application functioned correctly in the environment when compiled to an FPGA, it was far more likely to be correct once fabricated. The main drawback of the FPGA during this era was the performance. The FPGA was far behind the ASIC in terms of the most important performance aspects, such as latency, power consumption, etc. An application implemented on an FPGA was synthesized to the *static* nature of an ASIC and was not taking into account the dynamic reconfigurability allowed by the FPGA; in this sense, the performance of the FPGA can never overcome that of an ASIC [40]. However, as ASIC design has become increasingly expensive, and as the design rules have become significantly more complex, many of the tricks ASIC designers would use to squeeze all the performance out of their system have become too costly or error prone to be practical, thus narrowing the gap in performance between the two [45].

The power of reconfigurable systems lies in their flexibility. This flexibility not only allows for run-time circuit reorganization based on the input parameters, operating conditions, and updates, it also allows for a single circuit to be fabricated at incredible volumes. The high volume of FPGAs means that foundries are able to create them very cheaply (one shared mask, very large production runs, etc.) and very effectively (FPGAs are often the technology leaders, always available in the most aggressive lithography and processes, and carefully tuned to minimize variation and other deep submicron effects).

Due to the ability to customize the input data, many applications show speedups when implemented on reconfigurable systems. Many computing systems are fully reconfigurable at the logic level, and many devices are reconfigurable at the architectural level. In addition, reconfigurable cores are increasingly being used as components in embedded systems (e.g. microprocessors coupled with a reconfigurable component) as well as ASICs coupled with a reconfigurable component. Because of this, reconfigurable computing systems have emerged as an important organizational structure for implementing computations [9, 15, 19, 40, 51, 62]. They combine the generality of CPUs with the spatial computational style of hardware [19]. Reconfigurable systems use programmability and a regular fabric to reduce system complexity, cost, and development time.

1.2.2 The Internals of an FPGA

While there are many types of reconfigurable devices, FPGAs are the most common. FPGAs are configured by reprogramming logic at the *gate-level*, meaning that the device can be configured to look like any arbitrary set of interconnected logic gates. If a digital circuit schematic can be drawn for a design, then an FPGA is capable of emulating it. Physically, an FPGA is a collection of programmable gates embedded in a flexible interconnect, as shown in Fig. 1.1. These gates are implemented using lookup tables (LUTs) for computational units, flip-flops for timing, switchable interconnect for routing, and I/O blocks (IOB) for transferring data into and out of the device. Since any logic gate has a corresponding truth table, a circuit can be mapped to an FPGA by configuring the LUTs with the appropriate truth tables and by configuring bits in the switchboxes that specify which wires should be connected using pass transistors.[1] The data specifying how the LUTs and switchboxes should be programmed is referred to as a *configuration bitstream*.

The configuration of the FPGA may be stored in any number of ways. For example, FPGAs may use EPROM/EEPROM or antifuses (which are write-once technologies, meaning that the fuses are set and can never be unset). While many architectures make use of these write-once technologies, most architectures use SRAM as a programming point. The SRAM makes the FPGA volatile, meaning that it must be programmed every time that it is started up. Most importantly, SRAM allows for reconfiguration, which is the essence of reconfigurable computing. The SRAM programming bits are distributed across the entire FPGA, stored locally with the LUTs and switchable interconnects. While the large size of the internal configuration data may increase the amount of time required for reconfiguration, this has been partially mitigated through configuration caching and compression [33, 50].

Static RAM (SRAM) cells are built using two inverters and a couple of pass transistors (see Fig. 1.1). Data is stored or read from the SRAM cell as long as the

[1] A transistor in which an input is not only applied to the gate but also to the drain. This technology reduces the number of transistors required to implement certain kinds of logic.

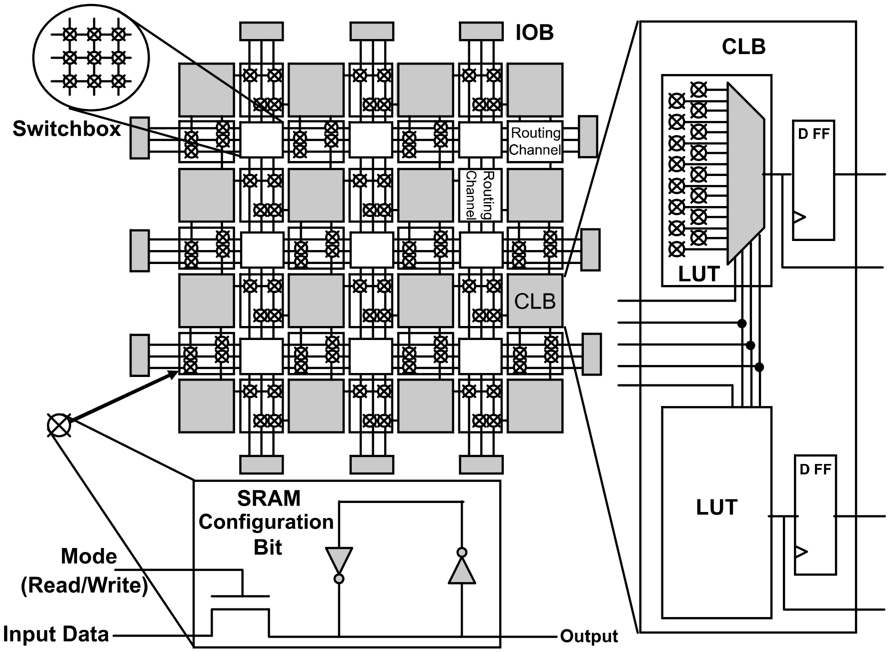
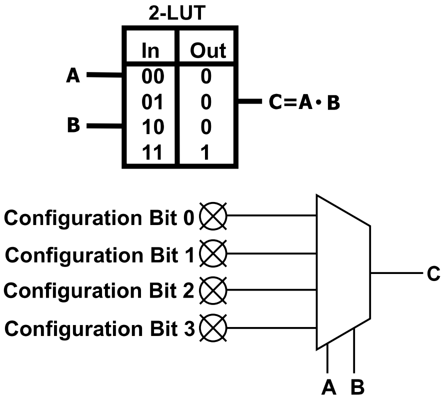


Fig. 1.1 Architecture of a typical FPGA. Multiple Configuration Logic Blocks (CLBs) are islands surrounded by a sea of interconnect. Each CLB contains a Lookup Table (LUT) that is configured to implement a primitive logic gate. The interconnect is also configurable, and it connects CLBs together so that more complicated circuits can be composed from the primitive logic gates. The FPGA bitstream specifies the configuration of both the CLBs and the interconnect

Fig. 1.2 This two-input LUT is configured to implement an AND gate



In	Out
00	0
01	0
10	0
11	1

cell is powered. Without power, the SRAM cell loses its value. LUTs use SRAM cells as programming bits. A LUT is very generic because it can implement any logic gate: an N-input LUT can implement any N-input function. Although 2^N bits are required to describe a LUT, it can implement 2^{2^N} possible functions. Figure 1.2

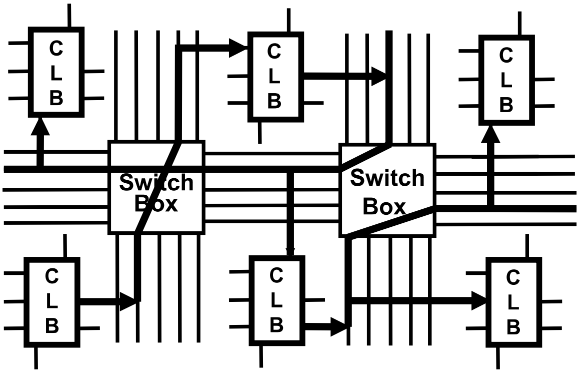
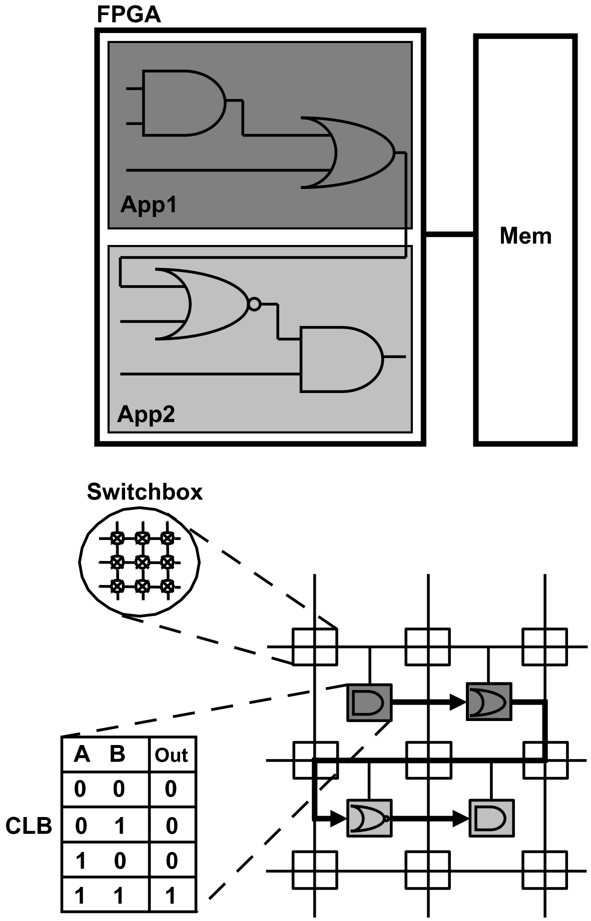


Fig. 1.3 FPGA interconnect architecture. Each CLB implements a primitive logic gate. More complex circuits are composed from primitive gates by connecting CLBs. The interconnect does the job of connecting the CLBs. Switch matrices are designed carefully because a full crossbar implementation would require N^2 connections. Implementing memory logic in reconfigurable hardware is inefficient because memories, which can take on any value, require switch matrices that are full crossbars. For this reason, FPGAs embed hard-wired memory and even processors among the reconfigurable logic

shows an example of a LUT. The size of SRAM cells limits the number of inputs that a LUT may have. LUTs typically have 4–5 inputs, based on extensive empirical work on optimal size and other aspects of FPGA architecture [7]. Modern FPGA architectures are organized into larger regions known as configurable logic blocks (CLBs). A CLB is a complex block consisting of LUTs, multiplexers, and flip-flops. In addition, these CLBs may contain custom logic to help the most common circuit patterns (e.g., the carry chains in ripple carry adders) increase the performance of these particular circuits. While in recent years these CLBs have continued to grow slowly, the granularity of these blocks is still much smaller than even the smallest microprocessor.

To connect the small computational blocks together, FPGAs employ an island style routing architecture. The generic island style architecture consists of two separate components for the routing architecture. The routing channel is a set of pass transistors that provide programmable connections into and out of the CLB. The switchbox provides point-to-point connections between neighboring routing channels, as shown in Figs. 1.3 and 1.4. Figure 1.5 shows how interconnect tracks are grouped into channels. Routing channels often contain *longlines*, which are used to span multiple CLBs in a row or column. The longlines create fast global connections between CLBs, which would otherwise have to pass through multiple, extremely slow switchboxes, as shown in Fig. 1.6. Figure 1.7 shows a switchbox with six pass transistors per switchbox interconnect point. A HARP switchbox is a different kind of switchbox that combines Hard-wired and Re-Programmable switches to increase routability [69]. Since resistance and capacitance increase quadratically in the length of the segment, longer routing segments are often driven by tri-state buffers, as shown in Fig. 1.8. The routing architecture is the major factor in both



A	B	Out
0	0	0
0	1	0
1	0	0
1	1	1

Fig. 1.4 Mapping the very simple circuit above to the FPGA fabric below. App1 consists of an AND gate and an OR gate, which are each mapped to a CLB. App2 consists of a NOR gate and an AND gate, which are each mapped to a CLB. The interconnect is configured such that the output of the AND gate of App1 is an input to the OR gate of App1; the output of the NOR gate of App2 is an input to the AND gate of App2; and the output of the OR gate of App1 is an input to the NOR gate of App2

delay and area of the FPGA. Approximately 90% of the area of a typical FPGA is used for interconnect (the space is required for both the physical wires themselves, and the configuration bits necessary to connect those wires together to emulate arbitrary interconnection networks). While this interconnect is critical to configurability, it also complicates the building of a secure reconfigurable infrastructure, as later chapters will describe.

An FPGA is programmed using a bitstream: in many ways the bitstream is analogous to a binary executable in a traditional microprocessor system. The FPGA is configured with this binary data to perform a particular function. The bitstream

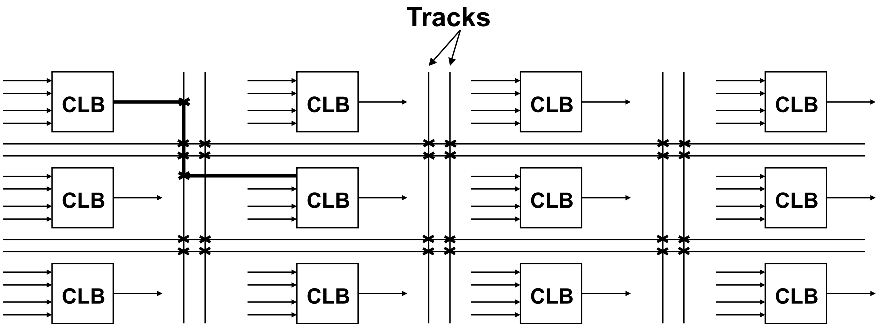


Fig. 1.5 Each logic element outputs one data bit, and the interconnect is programmable between elements. Interconnect tracks are grouped into channels

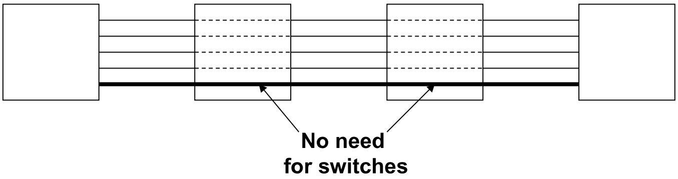


Fig. 1.6 Long segments only need switchbox connections at the end, therefore reducing the number of switches and therefore area

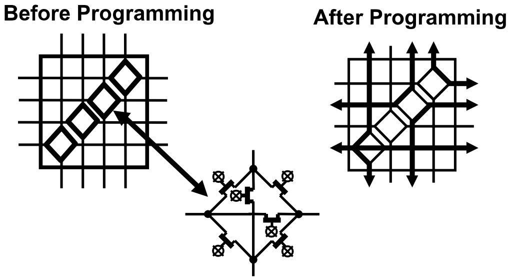


Fig. 1.7 A switchbox with six pass transistors per switchbox interconnect point. Pass transistors act as programmable switches, and pass transistor gates are driven by configuration memory cells. To implement a full crossbar would have required sixteen pass transistors

holds all parameters used to configure the FPGA, including the connections for the switchboxes and the data for the LUTs. Every bit in the bitstream corresponds to an SRAM configuration bit on the FPGA. To get a feel for the size of a typical bitstream, consider the Xilinx XC2V6000. It requires roughly one million bits to program the LUTs, and fifteen million configuration bits that control the interconnect structure plus various control functions. Programming this device takes between one to three seconds, depending on the specific configuration interface (e.g., JTAG) and the internal clock cycle that the device supports, and programming can happen when the developers specify. Because the FPGA stores these bits in volatile memory, it is necessary to read this bitstream each and every time the device is powered up. As later chapters will describe, this is an attractive target for those looking to subvert the

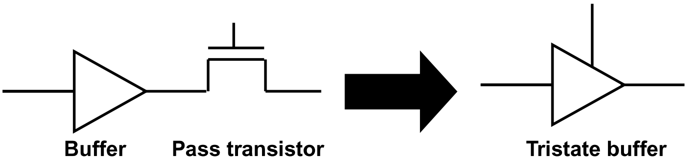


Fig. 1.8 In the design of the FPGA fabric itself, buffers are often used with longer segments because resistance and capacitance increase quadratically in the length of the segment

FPGA-based system. Modifications to this bitstream will allow arbitrary changes to the implemented circuitry, and if the bitstream can be read directly, an unscrupulous person can simply copy the bitstream into another FPGA, giving them the ability to clone hardware at a very low cost.

1.2.3 Design Flow

Usually when designing an FPGA-based system, the design flow is made to be very similar to that of a more traditional ASIC. The designs are typically developed in a hardware description language (such as VHDL or Verilog). These descriptions are then translated to a set of Boolean gates with the appropriate interconnect to route signals between them, at which point the design is typically tested with a variety of inputs. The CAD tools then translate these gates, packing them together into larger blocks, to logic functions which can map to the LUTs of the CLBs. The tools then determine the best place to locate these functional blocks across the chip (a step known simply as *placement*), and the routing between these blocks is calculated (known simply as *routing*). The design is then further analyzed to ensure that it is still correct and that it meets all the timing requirements. If there are problems with the functionality or timing, changes to the design or to the parameters given to the tools may be required. While this process has been used for a long time, ensuring correctness for large designs is a time-consuming endeavor. As such, most designs rely heavily on proprietary, *intellectual property* (IP) cores, which provide well-tested libraries that can be wired together to meet the demands of the specific application.

Code reuse, common in software engineering, is also used in designing reconfigurable systems in order to achieve cost savings and to reduce time to market. Designing a full system from scratch is not commonly done due to the high cost and development time. A commonly reused module is a soft CPU core, like the MicroBlaze. A soft-processor is a bitstream that implements a general-purpose processor's functionality on an FPGA. An example of composing a reconfigurable system would be to combine a soft-processor with other hard or soft IP cores, such as an AES core for crypto and an Ethernet core for networking.[2] The provenance of a core has im-

[2] Companies protect their intellectual property fiercely due to the high cost of designing hardware modules. Later chapters discuss several schemes to prevent the theft of IP cores from FPGAs.

portant security implications: was it developed in-house or purchased from a third party? Was it generated by a tool such as Base System Builder or downloaded from an open-source web site like opencores.org? While formal verification of crypto cores is a useful technique [49], determining whether an arbitrary hardware module is malicious is a non-computable problem, as it is with arbitrary software programs. For example, a core created by a malicious hardware designer or compromised design tool flow can exploit low-level hardware mechanisms to snoop on or disrupt system logic. Since a typical design contains millions of logic gates and ten times as many connections, designers need a way to compose trustworthy systems using multiple commodity cores on a single device without having to design each core from scratch (building everything from scratch is no guarantee of success either). Formal verification of the security of a large, complex design is a rigorous, expensive process, requiring access to the blueprints (e.g., HDL source) of all of the cores used in the design. Access to the HDL source might not be possible if it is a trade secret. Furthermore, formal verification is no silver bullet because of scalability issues. In addition, there can be a flaw in the mathematical model of the system, and there can even be a flaw in the mathematical proof applied to the model. Beware of claims that a system is *provably secure*. Flaws are frequently discovered in crypto ciphers that mathematicians “proved” to be secure [8].

Design Tip: *Tools and Cores*. An important aspect of reconfigurable system development is considering the provenance of design tools and IP, especially tools and cores downloaded from public web sites. It is a good idea to assess the security of development machines and code repositories. Later chapters will cover configuration control.

A typical FPGA-based embedded system consists of distinct cores residing on the same chip. Reconfigurable logic, hard-wired computational cores, and hard-wired SRAM and BRAM all reside on the same FPGA and can share the same off-chip memory. In many cases, cores must not be able to interfere with each other or snoop on each other via shared resources such as off-chip memory and on-chip memory and busses. As a result, the secure design of an FPGA system is challenging.

Figure 1.9 shows four possible embedded system design flows that can occur on the same FPGA:

- Logic synthesis transforms HDL code to a netlist, which is then converted to a bitstream by a process called *place and route*.
- An Electronic System Level (ESL) design tool like Celoxica transforms code written in C, a high-level programming language, to a soft processor core.
- Another ESL design flow uses AccelDSP [34] from Xilinx to convert MATLAB [52] algorithms to HDL, which can be converted to a custom DSP core.
- A C compiler translates code written in a high-level programming language to an executable, which is executed on a hard-wired processor core.

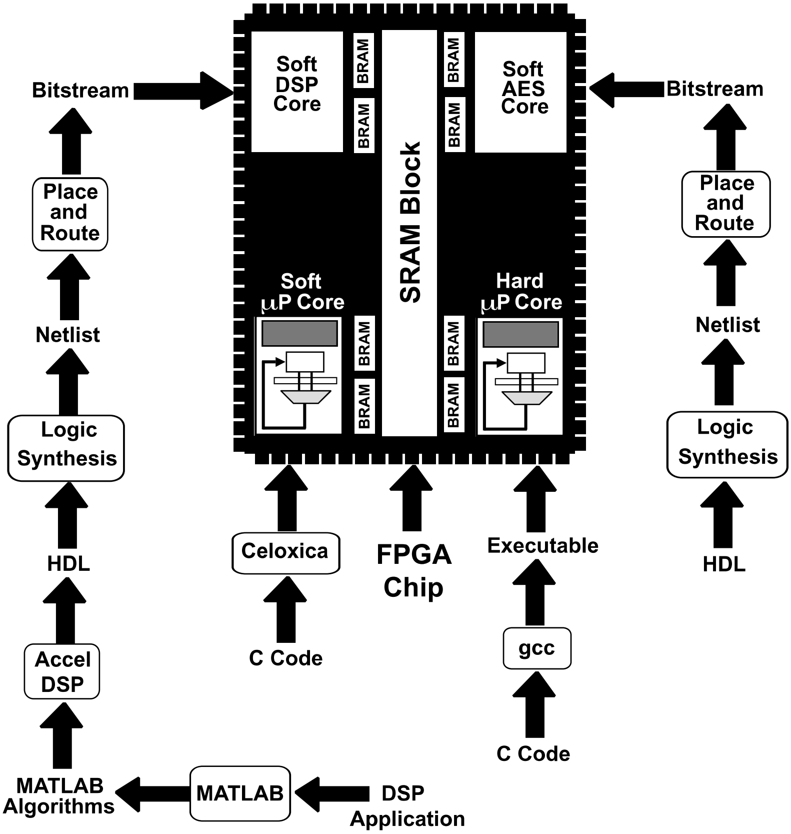


Fig. 1.9 A Modern FPGA-based Embedded System: Distinct cores with varying provenance reside on the same chip. Complex tool chains are used to generate cores. An AES core can be the result of transforming HDL code to a netlist to a bitstream. A DSP core can be result of transforming a DSP application to MATLAB algorithms to HDL to a netlist to a bitstream. This is an example of Electronic System Level (ESL) Design. Celoxica is another ESL design flow that transforms C code to a soft processor core. Finally, a C compiler such as gcc can transform C code to an executable, which can run on a hard-wired processor core

These tool chains represent difficult security vulnerabilities because tools are designed by many different companies and individuals. In addition to the provenance of tools, the provenance of soft IP cores, such as an AES crypto core, are also a major and challenging security concern. Cores can come in the form of HDL (e.g., Verilog), netlist,[3] or bitstream. They can be designed by a person or generated by tools.

The Xilinx ML507 evaluation platform is a real-world example of a development board that comes with a DVD of tools representing multiple types of design flows.

[3] A list of logical gates and their interconnections.

The ML507 has a Virtex-5 FXT FPGA with both reconfigurable logic and a hard-wired PowerPC processor core on the same chip, and Xilinx provides an Integrated Synthesis Environment (ISE) tool for generating bitstreams from HDL. The Embedded Development Kit (EDK) [78] tool is used to create a custom processor, allowing designers to connect their own hardware modules or other peripherals to a processor. The EDK also allows designers to configure the processor (e.g., by changing the frequency, cache size, etc.) The EDK also provides an integrated Eclipse-based Software Development Kit (SDK) for compiling and debugging the software to be run on the hard-wired PowerPC core or even on a soft MicroBlaze processor core.

Since these various design flows generate a design consisting of multiple interacting cores, the resulting embedded system may only be as trustworthy as the least trustworthy design flow, depending on the security architecture. For example, protection primitives are needed to isolate crypto cores so that secret keys cannot be compromised. Just as a subverted compiler can produce malicious code, subversion of hardware design tools can produce malicious hardware modules. Widely used design tools (e.g., Xilinx, Altera, and Actel) lack the ability to ensure that their output does not include malicious functionality. Furthermore, proving that a compiler or hardware design tool does not introduce malicious functionality is an open problem, at least as difficult as determining whether an arbitrary computer program or hardware module does something malicious.

1.3 The Many Facets of FPGA Security

As economics drives the growing use of FPGAs in critical systems, designers are forced to consider security, but practitioners currently lack a set of design practices to follow. In addition, providing security is made more difficult by the resource constraints in embedded systems [44]. In addition to applications of reconfigurable devices to security processing, researchers are beginning to think about the security of reconfigurable systems themselves. Since hardware designs can be copied from fielded systems, industry has invested heavily in developing methods to protect their intellectual property [11, 42, 47] and to ensure that FPGAs can only be updated by authorized parties [1, 31, 32]. However, only a few researchers have thought about malicious hardware modules on FPGAs [30]. Both Thomas Wollinger and Saar Drimer have written excellent, thorough survey papers on the subject of FPGA design security [22, 76]. Our survey papers were published in [35, 39]. Chapter 4 discusses bitstream encryption and authentication in greater detail.

A variety of attacks against FPGAs are possible. In a covert channel attack, a shared resource is used as a means of illicit communication. For example, power consumption can be modulated by a malicious core in order to send secrets to another core that observes these fluctuations [73]. Some FPGAs support remote updates of the bit-stream, and it is essential that only authorized parties be allowed to apply these updates. Otherwise, attackers could upload malicious logic that modifies the system's behavior or damages the chip by configuring the FPGA with a short-circuit [30]. While encryption [11, 41, 42], fingerprinting, [46], and watermarking [47] help to prevent IP theft, more is needed to counter covert channel attacks, side channel attacks, and to understand malicious hardware.

1.3.1 Security Is Hard

Computer security is a hard problem. Every year, the number of attacks increases despite increased spending on security. Commodity operating systems typically use a frustrating penetrate-and-patch approach, an endless cycle in which hackers compromise machines, vendors release a patch, and the hackers find and exploit new vulnerabilities. Ideally, systems should be built to be secure from the beginning, but designing a large, complex, highly trustworthy system is extremely challenging. There are no silver bullets: each security technique has its unique advantages and disadvantages. A principle-driven design and implementation strategy uses multiple complementary security concepts and techniques in concert. The goal is to increase the risk and cost for the adversary and to make attackers work hard to compromise each machine individually. Otherwise, systems can fall like dominoes once one is compromised. Relying on a single technique is akin to a Maginot Line of defense: attackers will simply take the path of least resistance to go around the Line. Systems that overly rely on the notion of a single method to create a security perimeter will encounter serious problems when the perimeter is violated. System designers should also beware of assuming that resources are safe within the perimeter.

Cryptography is a prime example of a security technique with advantages and limitations. Using crypto does not necessarily make a system secure: crypto needs to be used *properly*. Key length needs to be sufficiently long to resist a brute-force attack. Crypto processors must be designed carefully so that attackers cannot easily determine the key using a side channel attack. Keys need to be managed properly and stored securely. Ciphertext and plaintext must never mix. Flaws are occasionally found in the crypto algorithms, despite mathematical proofs of their security.

Design Tip: *Apply Comprehensive Security Principles*. Don’t rely on a single security strategy. Each technique has advantages and limitations. Don’t assume that attackers will not be able to breach arbitrary security perimeters. Just because a system uses crypto doesn’t mean it’s secure: keys need to be managed properly, keys need to have sufficient entropy, and the crypto mechanisms must be implemented correctly. The history of code making and code breaking tells us that there is no such thing as an unbreakable cipher. Keys that are sufficiently long today may not be sufficient in the future. Don’t assume that just because data is encrypted that it will never be decrypted in the future. Finally, don’t assume that you can protect your system by combining several weak security mechanisms.

1.3.2 Complexity and Abstraction

One reason that security is very difficult is the size and complexity of systems. Formal security analysis works best when applied to small, simple components. As complexity increases, so does the effort required for security analysis. This growth in complexity can be exponential in the size of the target of analysis. Furthermore, two components that are secure independently might not be secure when combined. One approach to get a grip on this complexity is to use abstraction. Another way to manage complexity is to limit what the user can do. For example, an ATM has a limited interface with a small number of buttons: every possible combination of keystrokes can be analyzed to determine whether they lead to an insecure state. Even so, ATM security has been repeatedly breached in the past [3].

A holistic approach to system security considers how the FPGA fits into a larger system. A large, complex system is composed at several different layers of abstraction: the chip level includes the components on a single FPGA device, and the board level includes the chips soldered onto a circuit board. Larger systems (e.g., a personal computer) can be composed of multiple circuit boards connected by a motherboard, which in turn can be connected by a computer network (e.g., Ethernet) to form systems-of-systems. The *composition problem* refers to the fact that two systems that behave in a secure manner by themselves do not necessarily behave securely when put together. This is especially frustrating for large design projects that must build systems from both trusted components and commercial off-the-shelf (COTS) components: reuse of commodity components is an important aspect of building large, complex systems. A *security architecture* is a technique for reasoning at a high level of abstraction about the composition of system elements, including computational and security components. A system can be designed to enforce a security policy by organizing the components using a security architecture, configuring the security mechanisms properly, and ensuring that all components conform to a well-defined interface.

To illustrate the difference between chip level, board level, and network level, take the example of the Xilinx ML507 evaluation platform, a development board in which CPUs interact with reconfigurable hardware. This development board has a Virtex-V FXT FPGA with a hard-wired PowerPC processor together with reconfigurable logic. Software running on the PowerPC interacts with soft processor cores implemented in reconfigurable hardware, all on a single chip. The FPGA chip interacts with other chips and components on the board, including DRAM, push buttons, LEDs, an LCD screen, USB, RS-232, Ethernet, DVI, etc. System security analysis must take into account all these interactions, at the chip level, at the board level, and at the network level (the ML507 can communicate to other computers via Ethernet and can even implement a web server). For example, it is possible to boot BlueCat Linux on the ML507, either running on the hard-wired PowerPC or on a soft MicroBlaze processor core in reconfigurable hardware. A flaw in the OS, application software, or MicroBlaze soft processor could be exploited by an attacker.

The reference monitor concept, discussed in Chap. 2, is another useful abstraction. The abstraction describes a perfect security mechanism that is a small, self-protecting, continuously invoked, security-critical piece of a system. It decides whether subjects (a.k.a. principals) in the system can access resources, according to a policy. Its small size allows the application of formal and complete security analysis. It is non-bypassable: no subject can get around it, and there is a well-defined interface to the reference monitor that all subjects must obey. Finally, it is tamper-proof: subjects must not be able to interfere with its functionality. This is why the reference monitor must be self-protecting.

Unfortunately, even formal methods are not a cure-all. The formal verification of secure systems requires the specification of a mathematical model of the system, and all proofs are constructed around this model. If there is a flaw in the model or a flaw in the proof, the system will not be secure. Formal verification is just one tool in the arsenal of designers of secure systems. Furthermore, the translation between formalization and implementation poses a problem.

Design Tip: *Security Architecture*. To manage complexity, develop a security architecture for your design that describes the organization of computational and security components. Think about the policy that you want your security architecture to enforce. Protect the security-critical parts of the design. Think about how the FPGA fits in with the rest of the system and how it interacts with other components. Beware of absolute claims that a system is secure because it uses cryptography or formal methods like theorem proving or model checking.

1.3.3 Baked in Versus Tacked on

Building a "high assurance" system that is resistant to attack by determined adversaries is extremely challenging and expensive. Security must be considered during all phases of the system lifecycle (i.e., "baked in") rather than an afterthought (i.e., "tacked on"). Lifecycle includes specification, design, implementation, configuration, operation, and updates. Security policies must be correctly specified. Security techniques must be easy for developers to use, and security mechanisms must be easy for end users to operate. Users must be trained in the correct operation of the system. System components must be engineered correctly. Design tools, hardware intellectual property, and software libraries must not be compromised.

Design Tip: *Build it in*. Consider security throughout all phases of the product cycle, including design, implementation, configuration, operation, and updates. You cannot tack it on at the end. Beware of claims that a fielded system is secure because it underwent penetration testing, a step that typically occurs at the end of the development cycle.

Despite the enormous challenges, people have known how to build highly trustworthy systems for decades. The Multics system, a predecessor of Unix, used several features to avoid common vulnerabilities such as buffer overflows which are still a problem today [38]. These features included a high-level programming language called PL/I, hardware permission bits, segmented virtual addresses, stacks that grow in the positive direction, and a rigorous development methodology. It was designed for mainframe computers, and it predated today's personal computers which operate in a hostile network environment. Although Multics wasn't perfect, today's operating system vendors have yet to apply many of its lessons, and trustworthy operating systems are not widely available.

Even in early time-sharing computers, users were uncomfortable with committing sensitive information to the care of a system in which there was little assurance of correct functional behavior [63]. This unease helped to clarify the need to understand exactly what these machines were doing and led to the articulation of principles and techniques for the construction of high assurance systems.

1.3.4 Separation of FPGA Cores

Separation is a foundational computer security concept that combines the ideas of isolation and controlled sharing. Cryptographic systems such as encryption devices were among the first to necessitate the development of strong isolation, since classified plaintext which we are envisioning being carried over *red* wires must be separated from ciphertext carried over *black* wires. Saltzer and Schroeder define complete isolation as a "protection system that separates principals into compartments between which no flow of information or control is possible" [61]. Since systems have limited functionality if all of their compartments are totally isolated, a technique is needed to facilitate the controlled sharing of data among components.

When FPGA modules are mapped to a physical device, their logic and interconnections may overlap significantly. This creates the possibility that an adversary may craft a module that intercepts or even corrupts *intra*-module communication in the same way that a network card can intercept Ethernet traffic. Even worse, the device can be physically destroyed if short-circuit configurations are downloaded. To securely manage an FPGA, the modules must be compartmented by placing them in distinct spatial regions on the chip. Establishing compartments also requires that interconnections between modules (e.g., busses) are carefully designed.

Moore's law has resulted in chips with one billion transistors, enough to fit hundreds of RISC processors. Computer architects cannot make uniprocessors any bigger without running into fundamental limits, and industry has placed its hopes (and fortunes) in multi-core architectures. Greater integration is also the reality for reconfigurable systems, and some FPGAs can fit as many as eight full-blown PowerPC soft processor cores. Many FPGAs have hard-wired processor cores in addition to the reconfigurable logic that can implement application-specific functionality. Reconfigurable systems-on-a-chip (SoCs) also have hard-wired multiplier units and blocks of SRAM and BRAM memory because certain circuits, especially memory, are much less efficient when implemented in reconfigurable hardware than when they are hard-wired. System complexity makes it challenging to provide security for these embedded systems. It is essential to isolate the system components to prevent a vulnerability in one component from threatening the entire system. For this, engineers of highly integrated reconfigurable systems need protection primitives for isolating the components while allowing components to interact and communicate in a controlled manner.

Managing shared resources such as off-chip memory is crucial for ensuring separation. While a CPU in a general-purpose system uses virtual memory to provide memory protection, FPGAs typically have a flat memory hierarchy and lack operating system support for virtual memory. Without these mechanisms, hardware modules can read and modify the memory of other modules. Furthermore, even if standard memory protection primitives were available, their designs generally assume the presence of an OS to ensure that the primitives are used securely. This situation calls for some way of enforcing a security policy on the embedded system. Chapter 5 presents a method that builds a mechanism in reconfigurable hardware to enforce a *memory access policy* that specifies the allowed sharing of memory between cores of an FPGA, a technique that is applicable to both on-chip and off-chip memory. Chapter 6 presents a method that exploits the spatial nature of FPGAs to enforce a policy that specifies the separation of cores: both the isolation of cores as well as their controlled interaction. Unlike the ASIC world, where incorporating novel security mechanisms is very costly (e.g., industry is often reluctant to incorporate security techniques devised by academics into commercial processor designs), it is easy to incorporate security primitives into FPGA designs. A wide variety of security mechanisms can be built on FPGAs, and the reconfigurable nature of FPGAs can be seen as both an asset and liability with respect to security.

1.4 Organization of This Book

The purpose of this book is to provide a practical approach to managing security in FPGA designs for researchers and practitioners in the electronic design automation (EDA) and FPGA communities, including corporations, industrial and government research labs, and academics. This book combines theoretical underpinnings with a practical design approach and worked examples for combating real world threats. To address the spectrum of lifecycle and operational threats against FPGA systems, a holistic view of FPGA security is presented, from formal top level specification to low level policy enforcement mechanisms. This perspective integrates recent advances in the fields of computer security theory, languages, compilers, and hardware. The net effect is a diverse set of static and runtime techniques that, working in cooperation, facilitate the composition of robust, dependable, and trustworthy systems using commodity components.

Chapter 2 begins with the fundamentals of security by presenting lessons from decades of computer security research in the high assurance software domain. Chapter 3 explains hardware security challenges, including malicious hardware, foundry trust, physical attacks, and the detection and mitigation of a specific type of covert storage channels on FPGAs. Chapter 4 describes the security issues related to dynamic partial reconfiguration of FPGAs and techniques for managing security in designs that employ partial reconfiguration. Chapter 5 explains a memory protection technique for preventing off-chip memory from being used to facilitate illegal information flow between FPGA cores. Chapter 6 explains a spatial separation technique for preventing FPGA cores from interfering with each other and describes a method for preventing illegal information flow between FPGA cores via the bus and direct connections. Chapter 7 provides a design example that incorporates the security primitives from earlier chapters. Finally, Chapter 8 describes forward-looking problems.

References

1. A. Abraham, It is I: an authentication system for a reconfigurable radio. M.S. thesis, Virginia Tech, August 2002
2. Actel Corporation, FPGAs for military, avionics, and high-reliability applications. *White Paper*, Actel Corporation, 2008
3. R. Anderson, Why cryptosystems fail, in *Proceedings of the 1st ACM Conference on Computer and Communications Security*, Fairfax, VA, November 1993, pp. 215–227
4. M. Attig, S. Dharmapurikar, J. Lockwood, Implementation results of bloom filters for string matching, in *Proceedings of the Field-Programmable Custom Computing Machines, 12th Annual IEEE Symposium on (FCCM'04)* (IEEE Comput. Soc., Los Alamitos, 2004), pp. 322–323
5. Z.K. Baker, V.K. Prasanna, A methodology for synthesis of efficient intrusion detection systems on FPGAs, in *Proceedings of the Field-Programmable Custom Computing Machines, 12th Annual IEEE Symposium on (FCCM'04)* (IEEE Comput. Soc., Los Alamitos, 2004), pp. 135–144
6. Z.K. Baker, V.K. Prasanna, Time and area efficient pattern matching on FPGAs, in *Proceeding of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays* (ACM, New York, 2004), pp. 223–232
7. V. Betz, J.S. Rose, A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs* (Kluwer Academic, Dordrecht, 1999)
8. A. Biryukov, O. Dunkelman, N. Keller, D. Khovratovich, A. Shamir, Key recovery attacks of practical complexity on AES variants with up to 10 rounds. IARC ePrint Report 2009/374, August 2009
9. K. Bondalapati, V.K. Prasanna, Reconfigurable computing systems. Proc. IEEE **90**(7), 1201–1217 (2002)
10. U. Bondhugula, A. Devulapalli, J. Fernando, P. Wyckoff, P. Sadayappan, Parallel FPGA-based all-pairs shortest-paths in a directed graph, in *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)*, April 2006
11. L. Bossuet, G. Gogniat, W. Burleson, Dynamically configurable security for SRAM FPGA bitstreams, in *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, Santa Fe, NM, April 2004
12. D.A. Buell, K.L. Pocek, Custom computing machines: an introduction. J. Supercomput. **9**(3), 219–29 (1995)
13. Y.H. Cho, S. Navab, W.H. Mangione-Smith, Specialized hardware for deep network packet filtering, in *12th International Conference on Field-Programmable Logic and Applications*, 2002

14. C.R. Clark, D.E. Schimmel, Efficient reconfigurable logic circuits for matching complex network intrusion detection patterns, in *Proceedings of FPL*, Lisbon, Portugal, 2003
15. K. Compton, S. Hauck, Reconfigurable computing: a survey of systems and software. ACM Comput. Surv. (CSUR) **34**(2), 171–210 (2002)
16. S. Craven, P. Athanas, Examining the viability of FPGA supercomputing. EURASIP J. Embed. Syst. **2007**(1) (2007)
17. J. Deepakumara, H.M. Heys, R. Venkatesan, FPGA implementation of MD5 hash algorithm, in *Canadian Conference on Electrical and Computer Engineering*, 2001
18. A. DeHon, Comparing computing machines, in *SPIE-Int. Soc. Opt. Eng. Proceedings of SPIE—the International Society for Optical Engineering*, vol. 3526, pp. 124–33, 1998
19. A. DeHon, J. Wawrzynek, Reconfigurable computing: what, why, and implications for design automation, in *Proceedings of the 36th ACM/IEEE Conference on Design Automation*, New Orleans, LA, June 1999
20. Design and Reuse Magazine, TTP controller IP in Altera's low-cost cyclone FPGA Families for Aerospace Applications, in *Design and Reuse Magazine*, 23 October 2007
21. S. Dharmapurikar, M. Attig, J. Lockwood, Deep packet inspection using parallel bloom filters. IEEE Micro **24**(1), 52–61 (2004)
22. S. Drimer, Volatile FPGA design security: a survey. Unpublished, Cambridge University, April 2008
23. Electronic Design Magazine, Actel FPGAs in Mars Rover, in *Electronic Design Magazine*, 6 August 2007
24. G. Estrin, Reconfigurable computer origins: the UCLA fixed-plus-variable $(F + V)$ structure computer. IEEE Ann. Hist. Comput. **24**(4), 3–9 (2002)
25. O.D. Fidanci, D. Poznanovic, K. Gaj, T. El-Ghazawi, N. Alxeandridis, Performance and overhead in a hybrid reconfigurable computer, in *Proceedings of the 2003 International Parallel and Distributed Processing Symposium (IPDPS)*, Nice, France, April 2003
26. M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, V. Hogsett, Granidt: towards gigabit rate network intrusion detection technology, in *Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications* (Springer, Berlin, 2002), pp. 404–413
27. J. Goodman, A.P. Chandrakasan, An energy-efficient reconfigurable public-key cryptography processor. IEEE J. Solid-State Circuits **36**(11), 1808–1820 (2001)
28. S. Govindavajhala, A. Appel, Using memory errors to attack a virtual machine, in *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2003
29. C. Grabbe, M. Bednara, J. von zur Gathen, J. Shokrollahi, J. Teich, A high performance VLIW processor for finite field arithmetic, in *Proceedings of the International Parallel and Distributed Processing Symposium*, 2003
30. I. Hadzic, S. Udani, J. Smith, FPGA viruses, in *Proceedings of the Ninth International Workshop on Field-Programmable Logic and Applications (FPL'99)*, Glasgow, UK, August 1999
31. S. Harper, P. Athanas, A security policy based upon hardware encryption, in *Proceedings of the 37th Hawaii International Conference on System Sciences*, 2004
32. S. Harper, R. Fong, P. Athanas, A versatile framework for FPGA field updates: an application of partial self-reconfiguration, in *Proceedings of the 14th IEEE International Workshop on Rapid System Prototyping*, June 2003
33. S. Hauck, L. Zhiyuan, E. Schwabe, Configuration compression for the Xilinx XC6200 FPGA, in *Proceedings of Symposium on FPGAs for Custom Computing Machines*, 1998, pp. 138–46
34. T. Hill, AccelDSP synthesis tool floating-point to fixed-point conversion of MATLAB algorithms targeting FPGAs. *White Paper*, Xilinx Inc., San Jose, CA, April 2006
35. T. Huffmire, B. Brotherton, T. Sherwood, R. Kastner, T. Levin, T. Nguyen, C. Irvine, Managing security in FPGA-based embedded systems. IEEE Des. Test Comput. **25**(6), 590–598 (2008)
36. B.L. Hutchings, R. Franklin, D. Carver, Assisting network intrusion detection with reconfigurable hardware, in *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'02)* (IEEE Comput. Soc., Los Alamitos, 2002), p. 111

37. Y.K. Kang, D.W. Kim, T.W. Kwon, J.R. Choi, An efficient implementation of hash function processor for IPsec, in *Proceedings of the Third IEEE Asia-Pacific Conference on ASICs*, 2002
38. P.A. Karger, R.R. Schell, Multics security evaluation: vulnerability analysis. Tech. Rep. ESD-TR-74-193, vol. II, HQ Electronic Systems Division, Air Force Systems Command, Hanscom Field, Bedford, MA 01731, June 1974
39. R. Kastner, T. Huffmire, Threats and challenges in reconfigurable hardware security, in *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, Las Vegas, NV, July 2008, pp. 334–345
40. R. Kastner, A. Kaplan, M. Sarrafzadeh, *Synthesis Techniques and Optimizations for Reconfigurable Systems* (Kluwer Academic, Dordrecht, 2004)
41. T. Kean, Secure configuration of field programmable gate arrays, in *Proceedings of the 11th International Conference on Field Programmable Logic and Applications (FPL'01)*, Belfast, UK, August 2001
42. T. Kean, Cryptographic rights management of FPGA intellectual property cores, in *Tenth ACM International Symposium on Field-Programmable Gate Arrays (FPGA'02)*, Monterey, CA, February 2002
43. P. Kitsos, O. Koufopavlou, Efficient architecture and hardware implementation of the Whirlpool hash function. IEEE Trans. Consum. Electron. **50**(1), 208–313 (2004)
44. P. Kocher, R. Lee, G. McGraw, A. Raghunathan, S. Ravi, Security as a new dimension in embedded system design, in *Proceedings of the 41st Design Automation Conference (DAC'04)*, San Diego, CA, June 2004
45. I. Kuon, J. Rose, Measuring the Gap between FPGAs and ASICs, in *Proceedings of the International Symposium on FPGAs*, Monterey, CA, February 2006
46. J. Lach, W. Mangione-Smith, M. Potkonjak, FPGA fingerprinting techniques for protecting intellectual property, in *Proceedings of the 1999 IEEE Custom Integrated Circuits Conference*, San Diego, CA, May 1999
47. J. Lach, W. Mangione-Smith, M. Potkonjak, Robust FPGA intellectual property protection through multiple small watermarks, in *Proceedings of the 36th ACM/IEEE Conference on Design Automation (DAC'99)*, New Orleans, LA, June 1999
48. P.H.W. Leong, I.K.H. Leung, A microcoded elliptic curve processor using FPGA technology. IEEE Trans. VLSI Syst. **10**(5), 550–559 (2002)
49. J.R. Lewis, B. Martin, Cryptol: High assurance, retargetable crypto development and validation, in *IEEE Military Communications Conference (MILCOM)*, Boston, MA, October 2003
50. Z. Li, K. Compton, S. Hauck, Configuration caching management techniques for reconfigurable computing, in *Proceedings of Symposium on Field-Programmable Custom Computing Machines*, 2000, pp. 22–36
51. W.H. Mangione-Smith, B. Hutchings, D. Andrews, A. DeHon, C. Ebeling, R. Hartenstein, O. Mencer, J. Morris, K. Palem, V.K. Prasanna, H.A.E. Spaanenburg, Seeking solutions in configurable computing. Computer **30**(12), 38–43 (1997)
52. The Math Works Inc. MATLAB user's guide. *White Paper*, The Math Works Inc., Natick, MA, 2006
53. D. McGrath, Gartner Dataquest analyst gives ASIC, FPGA markets clean bill of health. *EE Times*, 13 June 2005
54. C. McIvor, M. McLoone, J.V. McCanny, Fast Montgomery modular multiplication and RSA cryptographic processor architectures, in *37th IEEE Asilomar Conference on Signals, Systems, and Computers*, 2003
55. M. McLoone, J.V. McCanny, A single-chip IPsec cryptographic processor, in *IEEE Workshop on Signal Processing Systems*, 2002
56. Military and Aerospace Electronics, F-35 Joint Strike Fighter uses Actel FPGAs for engine electronics, in *Military and Aerospace Electronics*, 1 September 2004
57. Military and Aerospace Electronics, FPGA processors keep Mars Rovers moving, in *Military and Aerospace Electronics*, 11 January 2005
58. K. Morris, Cray goes FPGA, in *FPGA and Structured ASIC Journal*, 5 April 2005

59. H. Ngo, R. Gottumukkal, V. Asari, A flexible and efficient hardware architecture for real-time face recognition based on Eigenface, in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, 2005
60. C. Paar, B. Chetwynd, T. Connor, S.Y. Deng, S. Marchang, An algorithm-agile cryptographic co-processor based on FPGAs, in *SPIE's Symposium on Voice, Video, and Data Communications*, 1999
61. J. Saltzer, M. Schroeder, The protection of information in computer systems. Commun. ACM **17**(7), 388–402 (1974)
62. P. Schaumont, I. Verbauwhede, K. Keutzer, M. Sarrafzadeh, A quick safari through the reconfiguration jungle, in *Proceedings of the Design Automation Conference*, 2001, pp. 172–177
63. R. Schell, Computer security: the Achilles heel of the electronic Air Force? Air Univ. Rev. **30**(2), 16–33 (1979)
64. G. Selimis, N. Sklavos, O. Koufopavlou, VLSI implementation of the keyed-hash message authentication code for the wireless application protocol, in *IEEE International Conference on Electronics, Circuits, and Systems*, 2003
65. A. Senior, S. Pankanti, A. Hampapur, L. Brown, Y.-L. Tian, A. Ekin, Blinkering surveillance: enabling video privacy through computer vision. Technical Report RC22886, IBM, 2003
66. Z. Shi, R.B. Lee, Bit permutation instructions for accelerating software cryptography, in *Proc. of the IEEE International Conference on Application-specific Systems, Architectures and Processors*, 2000
67. Silicon Graphics, Inc., Extraordinary acceleration of workflows with reconfigurable application-specific computing from SGI. *White Paper*, Silicon Graphics, Inc., 2004
68. Silicon Graphics Inc., SGI builds world's largest FPGA supercomputer, boosts nucleotide query performance by more than 900 times over 68-node cluster. *White Paper*, Silicon Graphics, Inc., 8 November 2007
69. S. Sivaswamy, G. Wang, C. Ababei, K. Bazargan, R. Kaster, E. Bozorgzadeh, HARP: Hard-wired routing pattern FPGAs, in *Proceedings of the International Symposium on FPGAs*, Monterey, CA, February 2005
70. N. Sklavos, O. Koufopavlou, On the hardware implementations of the SHA-2 (256, 384, 512) hash functions, in *Proceedings of IEEE International Symposium on Circuits and Systems*, 2003
71. M.C. Smith, J.S. Vetter, X. Liang, Accelerating scientific applications with the SRC-6 reconfigurable computer: methodologies and analysis, in *Proceedings of the 19th IEEE Parallel and Distributed Processing Symposium (IPDPS)*, Denver, CO, April 2005
72. I. Sourdis, D. Pnevmatikatos, Pre-decoded CAMs for efficient and high-speed NIDS pattern matching, in *Proceedings of the Field-Programmable Custom Computing Machines, 12th Annual IEEE Symposium on (FCCM'04)* (IEEE Comput. Soc., Los Alamitos, 2004), pp. 258–267
73. F. Standaert, L. Oldenzeel, D. Samyde, J. Quisquater, Power analysis of FPGAs: how practical is the attack? Field-Program. Logic Appl. **2778**(2003), 701–711 (2003)
74. K. Underwood, FPGAs vs. CPUs: trends in peak floating-point performance, in *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, Monterey, CA, February 2004
75. J.E. Vuillemin, P. Bertin, D. Roncin, M. Shand, H.H. Touati, P. Boucard, Programmable active memories: reconfigurable systems come of age. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **4**(1), 56–69 (1996)
76. T. Wollinger, J. Guajardo, C. Paar, Security on FPGAs: State-of-the-art implementations and attacks. ACM Trans. Embed. Comput. Syst. **3**(3), 534–574 (2004)
77. L. Wu, C. Weaver, T. Austin, Cryptomaniac: a fast flexible architecture for secure communication, in *International Symposium on Computer Architecture*, 2001
78. Xilinx Inc., Getting started with the Embedded Development Kit (EDK). *White Paper*, Xilinx Inc., San Jose, CA, 2006

Chapter 2
High Assurance Software Lessons and Techniques

Abstract To understand the principles needed to manage security in FPGA designs, this chapter presents lessons learned from the development of high assurance systems. These principles include risk assessment, threat models, policy enforcement, lifecycle management, assessment criteria, configuration control, and development environments.

2.1 Background

Since the early 1960s system developers have been concerned with problems caused by *unspecified functionality*. This can include errors introduced in the development process and extra features added by industrious engineers. Sometimes extra features are relatively benign. In other cases, the unspecified functionality is malicious.

Engineers tend to trust hardware more than software. Sometimes engineers assume the hardware to be trusted; however, most malware can be implemented in hardware. The objective of this chapter is to introduce some of the lessons learned about avoiding mistakes in system implementations.

2.2 Malicious Software

Malicious software is functionality intended to violate the security policy of the system. The taxonomy of malicious software and the vulnerabilities such software exploits is vast: a 2007 report from the Common Vulnerabilities and Exposures project listed 41 different system vulnerabilities ranging from weak authentication to cross-site scripting attacks [18]. The focus of this chapter is on high assurance software lessons learned, and the discussion will be limited to two types of malicious software: that which executes in unprivileged domains and that which executes in privileged domains.

T. Huffmire et al., *Handbook of FPGA Design Security*,
DOI 10.1007/978-90-481-9157-4_2, © Springer Science+Business Media B.V. 2010

2.2.1 Trojan Horses

School children know the story of the fall of Troy to the Greeks [37]. Despite nine years of fighting, the Greeks were unable to breach the walls of Troy and conquer the city. Odysseus devised a plan to give the Trojans a present and pretend to retreat. Taking their gift, a large wooden horse, within the city walls, the Trojans celebrated until the Greeks emerged from the horse to sack and burn the city. The Trojan Horse was the vehicle for violation of the Trojan security policy: no Greeks within the city walls. When the Trojans found the Horse, it was on the beach: it was the Trojans who dragged it into the city and then "activated" it by celebrating.

In terms of computer systems, a Trojan Horse is hidden functionality within software, where the latter provides some other desired service. So if a user downloads or otherwise installs an application or functionality of unknown provenance, a Trojan Horse may accompany it. When a Trojan Horse executes in the context of a user application, it is generally constrained by the privileges granted to that executable, which are derived from the privileges accorded the user. The adversary has no control over when the malware will execute: if the user never invokes the application that contains the Trojan Horse, then it may never execute.

Despite these constraints, in most systems a Trojan Horse can wreak havoc on confidentiality, integrity and availability objectives. Consider the situation illustrated in Fig. 2.1. Kathy has a file containing information that Sean should not access. She has set the access controls on her file so that Sean will not be able to access the information directly. Unfortunately, Kathy is executing an application that contains a Trojan Horse devised by Sean and his nefarious gang. Although the legitimate application may make legitimate use of Kathy's file, the Trojan Horse writes her information into Sean's file. She has no idea that this is occurring.

Modern Trojan Horses may not exhibit behavior as simplistic as that illustrated in Fig. 2.1. Instead they may send information to remote sites. The activities of these applications are often rather complex, and the mechanism used to transmit may not be visible to kernel-level auditing mechanisms.

As will be seen in Sect. 2.5.1.2, the activities of Trojan Horses can be confined when mandatory policies are enforced.

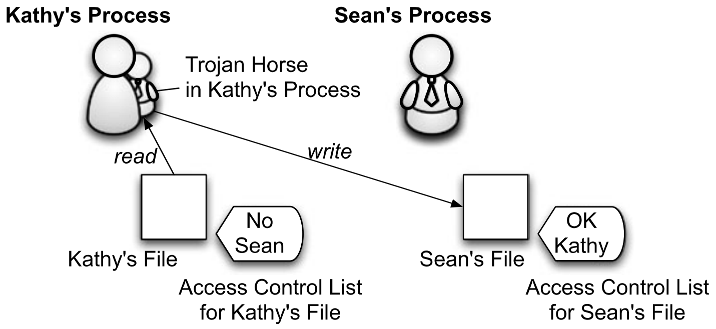


Fig. 2.1 Although Sean has no access to Kathy's file, the Trojan Horse in Kathy's process can write all of her information to Sean's file

2.2.2 Subversion

A well known example of subversion is the simple flight simulator in early versions of the Excel spreadsheet [100]. It was activated by a set of conditions that were highly unlikely to occur during typical use of the spreadsheet. It allowed the user to fly around over a gloomy landscape that featured a tombstone upon which credits scrolled, presumably containing the names of members of the development team. A less benign example of unspecified functionality is a hypothetical backdoor inserted into an operating system by its compiler [53, 101]. Anderson provides a worked example of system subversion [4]. Although no attempt is made to obfuscate the artifact, it consists of a total of eleven lines of code and allows the attacker undetected access to the entire Linux file system. Because the attacker may not know the exact nature of the ideal attack when a trapdoor is installed, it may be prudent to develop a chained subversion that consists of a toehold for subsequent system exploitation, a loader for putting the malicious payload in place, and the payload itself for the attack *du jour* [58, 72, 81].

Subversion differs from a Trojan horse in the following ways. First, a subversion can be activated by the adversary at will, whereas a Trojan horse requires the cooperation of the victim. A corollary is that the adversary can choose the time of activation, usually via triggered activation and deactivation, but, for a Trojan Horse, the time of activation depends upon the victim's use of the software. Third, a low-level subversion will bypass the security controls, and, in contrast, a Trojan horse will be constrained by the controls placed upon the victim executing it. Finally, Trojan Horses generally execute in the application domain, whereas the ideal execution domain for a subversion is the operating system.

Attackers can choose a number of system lifecycle phases, both developmental and operational, during which to target a subversion attack. The objective is to implant an artifice in the system that can execute with unlimited privileges. Myers identifies a number of lifecycle opportunities for subversion [74]. A system's lifecycle can be divided into three major stages: development, operation, and retirement. Developmental threats result in the incorrect construction of the system such that the high level policy and specifications are not faithfully reflected in the system implementation. Both unintentional errors and intentionally inserted or unspecified functionality fall into this category.

Operational threats can expose information assets to a variety of attacks. Imprecise interface specifications involving groundless assumptions can be exploited. Exploitable flaws may result from poor design and implementation. Chained attacks may allow attackers access to critical information. If the system is constructed in a way that permits its operational state to be manipulated, covert channels [60] may be possible. In the case of hardware, it may be possible to extract information using side channel attacks involving power or timing analysis, e.g. [56].

After further subdividing these phases, Myers identifies subversion threats for each phase and recommends rigorous design and development methodologies applied to people, processes and tools, accompanied by lifecycle assurance as a holistic approach to the mitigation of these threats. Table 2.1 extends this analysis to include system retirement.

Table 2.1 Lifecycle opportunities for subversion

Phase	Threat
Design	Inclusion by high-level designers of exploitable design elements
Implementation	Introduction of flaws and artifices in code base
Distribution	Additions to product and bogus updates
Installation	Untrusted installers insert artifices or misconfigure the system
Operation	Exploit flaws to install trap doors
Retirement	Analyze system and media to extract information

Subversion of an FPGA can occur at many levels ranging from developmental attacks on the hardware and software to operational attacks. Of course, the devices can be subverted at the IC level; however, given that the IC manufacturer does not know the use to which the base array will be applied, such attacks are probabilistic and lack the guarantees desired for a well designed subversion. As posited by both Karger and Schell [53] and Thompson [101], the tools used to construct the FPGA offer a vector for developmental subversion. Trimberger [102] describes some of the challenges associated with non-destructive validation of the equivalence between the original design and the implemented design. He recommends the use of layout-versus-schematic (LVS) comparison tools as a means to detect subversion. Despite the finite nature of the systems, such detection schemes are extremely challenging. Techniques that may be considered for countering subversion in FPGAs include: testing and validation of design tools, verification of design flows, and static analysis of HDL code.

2.3 Assurance

A system's *functional* security mechanisms are those that implement the access control rules, the login mechanism, the audit trail, and various security administrator functions. In contrast, *assurance* relates to the trustworthiness of the system. Just because a user might trust a system does not mean that it merits that trust: it must be trustworthy. This maps to a confidence that the system is doing what it is supposed to do and nothing extra. If the system contains errors or is implemented in a way that permits unintended use of its interfaces, then it contains unspecified functionality. In addition, a system might contain functionality that is intentionally inserted by a malicious adversary.

It is up to system owners to determine the adequate amount of assurance. The trustworthiness or assurance of systems is elevated through careful lifecycle management from the elicitation of requirements through retirement. At any step along the way, an adversary may try to enter the system to add some *special* functionality. Careful system security engineering, configuration management, trusted delivery mechanisms, testing, both internal and external reviews, and the application of formal methods contribute to system trustworthiness.

Design Tip: *Assurance Requirements*. Security is not for free. Spend your security dollars wisely. Your analysis should consider the assets that require protection and the resources available to the adversary. A higher level of trustworthiness requires greater effort to properly design, implement, test, deliver, configure, operate, and audit. While formal methods may be necessary for high assurance, they are not sufficient, in and of themselves.

2.4 Commensurate Protection

In early work with shared computer systems, when various conceptual models for computer processing (e.g., processes and scheduling) and information protection were formed, vendors presented different approaches as secure, and the computing community in ad hoc or organized [53] efforts would discover and report the incompleteness or incorrectness of the system's protection features. Today, the relationship between computer vendors and their customers remains largely the same, although many vendors may no longer assert that their systems are "fit" for security purposes [119], lest they be held liable for the product's adequacy [27]. A vendor releases a product; users discover and report some of its vulnerabilities; the vendors patch the vulnerabilities and add new features; and the vendor releases the product anew. The patches and new features, especially in combination, may add new vulnerabilities. While this *penetrate-and-patch* approach is not a satisfactory process, it avoids the up-front cost of building in security. Since vendors have found that users are willing to put up with vulnerable products, the cycle continues.

Security is expensive to build into a product, as it increases the design, documentation, configuration management, and testing efforts. While this careful approach to development (see Sect. 2.3) may reduce the *overall* cost of product maintenance, those long-term savings may not be persuasive to vendors who compete on a "first-to-market" basis. In any event, since security is not free, the question arises for data owners as to how much security is enough.

Common wisdom about the protection of any property is that one should not spend more on protection than the value of that which is protected. Another maxim is to not gamble (i.e., leave unprotected) that which you cannot afford to lose. Cost-benefit and risk analysis methods can be used to quantify the level of protection required based on the value of the information: i.e., the damage to the owner if the information is violated (compromised, corrupted, or made unavailable [61]). FEMA [50] uses the following generic formula to calculate financial risk, assuming that threat and vulnerability ratings range from 0 to 1:

$$Risk = Asset\ Value \times Threat\ Rating \times Vulnerability\ Rating$$

With respect to information protection, other risk factors may include the *proportion* of the Asset Value that will be lost if the information is violated, the value of the information to potential attackers (which may be different than the value to the owner), and mitigations to vulnerabilities. The combination of a computing system's vulnerabilities and its mitigations to those vulnerabilities can be viewed as the inverse of the *protection* it provides with respect to defined assets and threats. Various approaches have been presented for measuring the protection provided by IT systems, including different evaluation criteria [14, 110].

Assets protected by IT systems may include people, the valuation of which can include factors such as life and liberty, which may have a subjective relationship to their monetary value. The *threats* to assets can also be difficult to quantify, as discussed next.

2.4.1 Threat Model

Looking at the FEMA risk formula, enough protection (i.e., mitigation to vulnerability) must be provided to keep risk to the assets within acceptable limits, given the perceived or defined threats. If there is no threat (e.g., if one has assets that no one else wants to attack), or the protection system has no vulnerability (assets are protected completely and continuously), there is no risk. However, a conservative assumption is that attackers' motivation and resources are proportional to the value of the information resources: highly valued information requires high assurance of protection. A *threat model* provides a more systematic evaluation of threat [57, 66, 73], including factors such as:

- The nature of the asset—whether its value derives from its confidentiality, integrity, and/or availability; and if there is a temporal quality to the value (e.g., some intellectual property or strategic military data may need to be secured for decades).
- The potentially vulnerable components or interfaces of the system through which assets can be accessed; and the attacks known to be pertinent to each type of the component or interface throughout the product life-cycle (e.g., design, development, delivery, and maintenance).
- The adversary's motivations, including monetary, competitive advantage, industrial espionage, revenge, prestige, and notoriety. This can be related to the nature of the assets.
- The adversary's capabilities—technical expertise, access to the protected system, and the availability of funding and other resources such as computer time and exploitation tools.

In addition to the primary IT access control features, potential attack surfaces include the procedural as well as the automated instantiation of various design assumptions and *supporting policies*, including: identification, authentication, audit, physical security, and the education and vetting of users. For example, the most cost effective means of attacking a system could be through the use of social engineering and bribes.

A threat model is useful for the assessment of components and products as well as for system deployment environments, which can be more specific regarding the characterization of threats, assets and adversaries. The following relates threats to components and products.

2.4.1.1 FPGA Interfaces

For FPGA components, a threat model should consider both internal and external interfaces. Cores may be connected directly or by a bus. Network interfaces may include malicious traffic. The FPGA reconfiguration interface must be considered—e.g., some FPGAs can be remotely updated in the field—as well as interfaces to shared computing resources such as system and cache memory.

2.4.1.2 FPGA Assets

General classes of information assets on FPGAs include: cryptographic keys, private information, and proprietary logic designs.

2.4.1.3 FPGA Attacks

For systems that allow untrusted modules or applications to share the processing environment, it must be assumed that they are malicious. Attack analysis must include those attacks related to system design vulnerabilities (if any) that are described in the product security evaluation and the open literature. Other potential FPGA attacks include:

- Uploading a malicious design through the FPGA reconfiguration interface. A malicious design could actually melt parts of the FPGA by causing a short-circuit [41].
- Exploiting the effects of using or contending for shared resources. For example, if one core can measure the effects caused by the use of a computing resource (e.g., delay in access to cache memory, change in temperature of the device, or change in the use of electrical power) by another core, a covert channel or side channel can result.

2.4.1.4 Other Threat Model Elements

Physical attacks, e.g., to read or destroy cryptographic keys, may require tamper protection, detection, and response techniques at the system, board, or chip level, depending on the value of the assets protected. Also, FPGA manufacturing and development tools are an attack vector in which the tools are subverted to weaken the FPGA designs they produce, as a second order effect. Chapter 3 discusses physical attacks in greater detail.

2.5 Security Policy Enforcement

Security requires the specification of a policy for a system and the translation of that policy to a system implementation that enforces the policy.

2.5.1 Types of Policies

When constructing a secure system it is essential to establish what *secure* means. Assuming that resources managed by the system must be protected, it is necessary to understand the kinds of protection that should be established. Security is always understood with respect to a policy. *Information assurance* is generally defined as a set of measures intended to protect and defend information and information systems. The five pillars of information assurance provide a way to categorize overarching objectives:

- Confidentiality—information is accessible only to those who need it and protected from unauthorized disclosure.
- Integrity—information is modified only by those with appropriate authorization and is protected from corruption.
- Availability—information is usable when needed in a reliable and consistent form.
- Authenticity—the recipient of information has knowledge of its genuine sender or source.
- Non-repudiation—irrefutable evidence of message transmission from its sender and receipt by its receiver can be provided.

The first three can be associated with system resources, whereas the last two are related to communications and are supported by well-designed protocols and the enforcement of some combination of the first three. For this reason we will focus only on the first three objectives. Sterne [99] describes an *organizational* security policy as one that may be stated in very general terms. Its translation to a system implementation results in an *automated* security policy. The automated policy is usually a subset of the overall policy, since policies related to physical and personnel security are beyond the scope of the computer system implementations.

We understand IT systems to be *secure* in the context of the policies regarding the confidentiality, integrity, and availability of the information resources. Different system security policies may combine confidentiality, integrity, and availability in different ways. For example, in a real-time control system the availability of high-priority events will be paramount, and the confidentiality of information may be secondary. A system intended to manage corporate financial records may focus on integrity and associated accountability controls. Finally, a system designed to protect state secrets will ensure that confidentiality policies are enforced. Other examples include voting systems, health record systems, and employee payroll systems. Security engineers often find a tension in the CIA triad: all three policies cannot be perfectly enforced simultaneously! The existence of a tension between availability and both confidentiality and integrity is most evident, for example in military real-time systems that must manage both classified and unclassified information.

Before continuing with a more detailed discussion of policy enforcement, it is necessary to introduce certain terminology that has proved useful for several decades.

The underlying policy enforcement mechanism controls resources, and a subset of those resources will be exported at the mechanism interface in the form of abstract data types. These may include both active and passive entities. Those resources that can be read from or written to contain information and are called *objects*. The active entities in the system are called *subjects* and may be surrogates for typical users or for system owners. A typical example of the former might be an ordinary application process, and an example of the latter might be a service process [59].

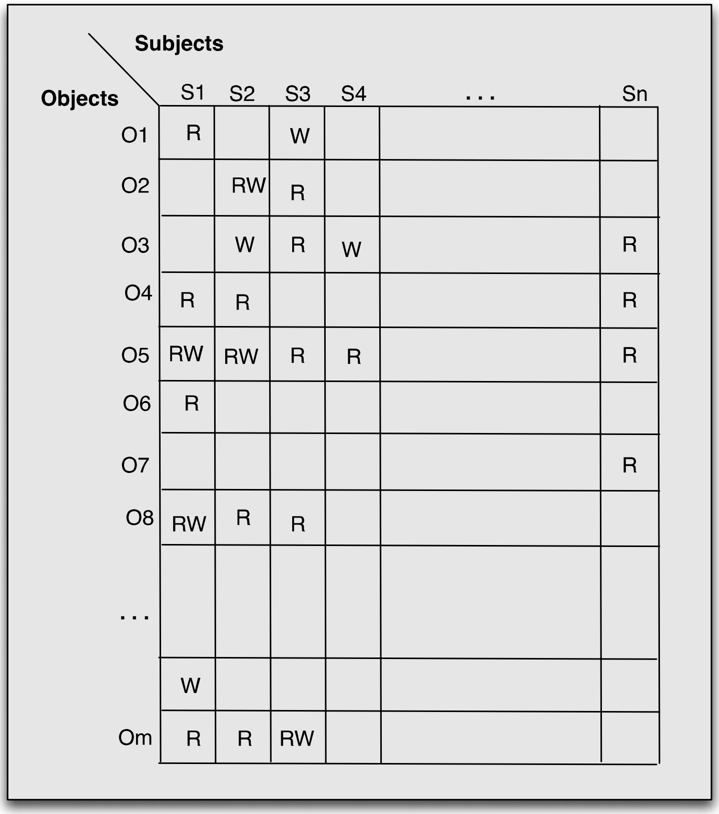
Early pioneers in system security developed models to characterize policy enforcement mechanisms. For example, Graham and Denning [40] developed a tabular model that described the rights of each subject to objects. As depicted in Fig. 2.2, each cell in the matrix contains the access modes with which the corresponding subject is permitted to access the corresponding object.

The parameters used in these checks come from policy-relevant metadata associated with both the subjects and the objects. The nature of the metadata will be determined by the kind of policy being enforced. Any policy can fall into one of two major categories: discretionary and mandatory. So, in some systems enforcing discretionary access controls, user names or groups may be bound to subjects, and some metadata, such as a list of allowed users, may be associated with the object. In a system enforcing mandatory access control policies, the metadata may consist of sensitivity labels associated with both the subjects and objects. The most familiar types of labels are those used by the military to classify information, e.g. CONFIDENTIAL, SECRET, etc.

An interesting consequence of access control systems with interfaces that permit policy modification is that it is impossible to develop an algorithm to decide for an arbitrary protection system whether or not information will leak in an unintended manner [42]. Considerable research has explored the precise characterization of protection models that *are* decidable [2, 86, 87].

2.5.1.1 Discretionary Policies

A discretionary policy is dynamic and can be modified by unprivileged subjects during runtime, whereas a mandatory policy is *immutable* to those subjects. A non-technical example of each can be found in sentencing guidelines of the criminal justice system. For many crimes, the presiding judge is able to weigh a variety of factors associated with a particular case and can determine a punishment that *fits the crime*. Alternatively, judicial discretion might be constrained through the passage of a variety of sentencing mandates, such as *three strikes* laws. Where such constraints are in place, the sentencing judge has no choice regarding the punishment: there is no discretion.



Objects \ Subjects	S1	S2	S3	S4	. . .	Sn
O1	R		W			
O2		RW	R			
O3		W	R	W		R
O4	R	R				R
O5	RW	RW	R	R		R
O6	R					
O7						R
O8	RW	R	R			
. . .						
	W					
Om	R	R	RW			

Fig. 2.2 The Graham-Denning model described access control in terms of a matrix where the rights of a subject to an object were given in the cell associated with the subject and object

In systems that enforce *discretionary policies*, an interface is provided that allows applications or users to modify the policy. Figure 2.3 illustrates how this can be a problem. Kathy's subject is executing a program that contains a Trojan Horse. As the Trojan Horse code is executed, Kathy's subject will use the runtime API to change the Access Control List (ACL) on her file to grant access to Sean. At this point, Sean's subject can read her information and store it for later use. Because the discretionary policy is *ad hoc*, it is impossible for Kathy to protect her information from access by Sean.

2.5.1.2 Mandatory Policies

In addition to being immutable, a *mandatory policy* is one that is both global and persistent: the policy is the same everywhere, and it does not change depending upon various conditions. If Jim's *secret barbeque sauce* is secret in Texas, it is also secret

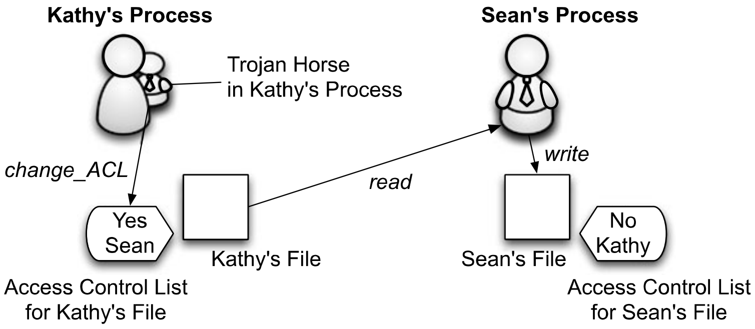

Kathy's Process
Sean's Process
Trojan Horse in Kathy's Process
change_ACL
write
read
Yes Sean
Kathy's File
Sean's File
No Kathy
Access Control List for Kathy's File
Access Control List for Sean's File

Fig. 2.3 A change to the ACL on Kathy's file permits Sean to read and store her information

in Berlin. In addition, Jim does not allow the recipe for the sauce to be available on Tuesdays from nine to ten o'clock in the morning: once the recipe is available, it is impossible to make it secret again. These global and persistent policies separate information and those who can access them into a lattice of partially ordered equivalence classes [30]. A well-understood mandatory policy is that of the military, which requires the classification of information based upon the harm its disclosure would cause to the nation. Typical military information classifications are: TOP-SECRET, SECRET, and UNCLASSIFIED. Individuals cleared for TOP-SECRET may access TOP-SECRET, SECRET, and UNCLASSIFIED; those cleared for SECRET may access SECRET and UNCLASSIFIED; and uncleared individuals may only read UNCLASSIFIED information. Because these classifications can be hierarchically ordered, this is called a multilevel security policy. However, as Denning pointed out, a partial ordering may have non-comparable equivalence classes, so sets of information may be organized in a MLS policy as well, as illustrated in Fig. 2.4. The arrows show the direction of information flow. Thus the reader must possess either a label of $\{b, c\}$ or $\{a, b, c\}$ to read information labeled b, c.

Two state machine models capture the intent of mandatory confidentiality and integrity policies respectively. The Bell and LaPadula model [7, 8] describes the secure state of a system and includes three properties: the *simple security property*, the **-property*, and the discretionary property. In this chapter, only the first two properties are of interest. If a system exhibits the simple security property, then an entity will only be able to read information at and below its confidentiality level. Another way of stating this is that it reflects typical confidentiality policies that prohibit individuals from reading information at a higher classification level than that for which they are cleared. The second property that must hold for the system is the *-property, which accidentally has this unfortunate name. It is often also referred to as the *confinement property* to reflect the notion of information confinement [60], and accounts for the challenge posed by Trojan Horses in user applications. As a result of confinement, it is impossible for an entity at a high confidentiality level to write to an information repository at a lower confidentiality level.

In the context of mandatory policies, integrity forms a dual of confidentiality. High integrity information should only be modified by high integrity entities,

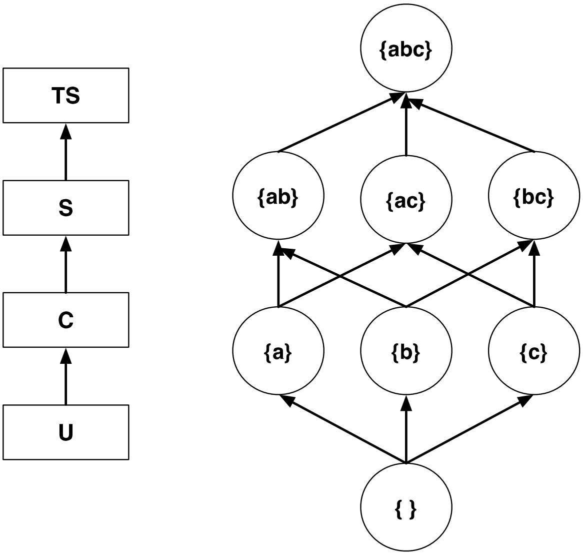


Fig. 2.4 A hierarchical ordering is shown in the lattice on *the left*, and a lattice of sets is shown on *the right*

whereas high integrity information should be accessible to entities at all integrity levels, even the lowest. Thus the Biba model [9] includes properties that constrain observation, modification and invocation.

2.5.1.3 Least Privilege and Its Policies

The *Principle of Least Privilege* is one of the cornerstones of secure system design, implementation, and management. It appeared in a codified form in the seminal paper by Saltzer and Schroeder [85] in which they described eight design principles that can guide the construction of secure systems. Their definition of least privilege stated that "every program and every user of the system should operate using the least set of privileges necessary to complete the job. Primarily, this principle limits the damage that can result from an accident or error. It also reduces the number of potential interactions among privileged programs to the minimum for correct operation, so that unintentional, unwanted, or improper uses of privilege are less likely to occur." Hardware mechanisms can create *protection modes* within a system that limit the privileges of applications with respect to those of the kernel (see Sect. 2.5.2.1).

Least privilege will be reflected in system design and implementation through the use of layering, modularity, and information hiding, all of which are constructive techniques that, when applied to the internal architecture of a system, improve the system's resistance to penetration. The system interface can export access control and fine-grained execution domains such that subjects may only perform authorized tasks.

2.5.2 Policy Enforcement Mechanisms

To enforce an access control policy, a mechanism must be in place that checks the access of the subjects to objects.

Since its introduction, the *Reference Monitor Concept* [3] has served as a useful abstract model for systems enforcing security policies. As an idealization of such systems, it can be used as a standard of perfection against which those designing protection mechanisms can measure their implementations.

The Reference Monitor Concept does not refer to any particular policy to be enforced by a system, nor does it address any particular implementation. Instead it articulates three properties of an ideal access mediation mechanism:

- The access mediation mechanism is always invoked: every access is mediated. If this were not the case, then it would be possible for an entity to bypass the mechanism and violate the policy.
- The access mediation mechanism is tamperproof. Thus, it is impossible for a penetrator to attack this ideal access mediation mechanism so as to disable the required access checks.
- The access mediation mechanism itself "must be small enough to be subject to analysis and tests, the completeness of which can be assured" [3]. This means that the mechanism must be understandable. It is necessary to ensure that it is doing what it is supposed to do and no more.

This articulation of a mechanism has met the test of time and continues to be an effective tool for describing the abstract requirements that drive secure system design and implementation. No viable alternative has been introduced, and it has proven effective, even under close scrutiny.

The minimal requirements for protecting the most privileged system elements from less privileged applications were described by Saltzer and Schroeder [85]. They include privilege bits, a memory management mechanism, controlled entry points to privileged functions, and a trusted way to bind user attributes to those of the active entities executing on behalf of the user. Each of these requirements will be discussed in greater detail in the next sections.

2.5.2.1 Privileged Instructions, Rings, and Gates

Systems are organized in terms of privilege. Hardware resources are managed by the most privileged software components, which organize and export abstract data types at an interface used by the next less privileged components. The simplest privilege hierarchy is that of a two-state processor that provides two privilege domains. The *privileged* domain is used by the operating system or kernel, and applications occupy the *unprivileged* domain. More elaborate hardware architectures support several hierarchical privilege domains, such as those of the Intel x86 family of processors, which has four hardware privilege levels [47].

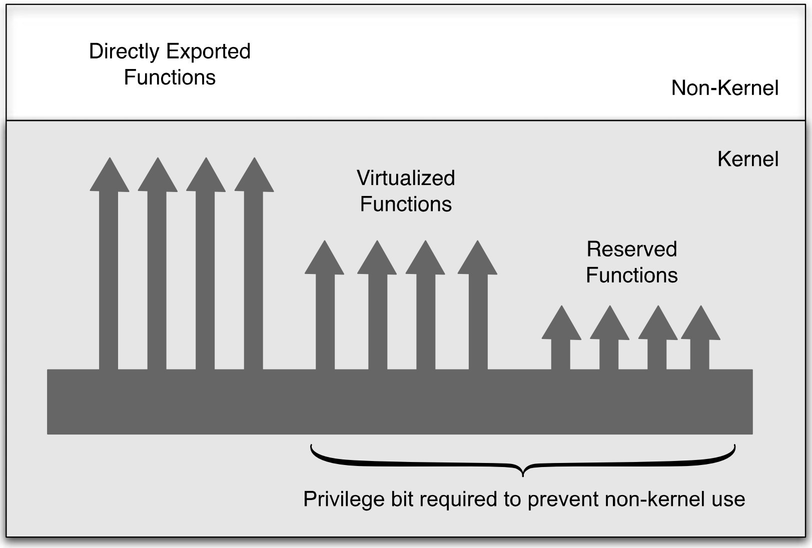

Directly Exported Functions
Non-Kernel
Kernel
Virtualized Functions
Reserved Functions
Privilege bit required to prevent non-kernel use

Fig. 2.5 Hardware instructions may be exported directly at the kernel interface, virtualized and exported as new abstract data types, or reserved for the exclusive use of the kernel

The ability of the processor to check the privilege level of an active entity within a task when attempts are made to access resources, transfer to a different privilege domain, or execute selected instructions, is an essential element of the overall protection mechanism provided by the hardware. *Privileged instructions* will only be executable by entities in the most privileged domain; if an application attempts to execute a privileged instruction, the hardware will issue a protection exception, and processing may be forced into an exception handler. Instructions that affect the state of the processor such as those to manage hardware memory resources, manage control and other registers, halt the processor, and perform a limited number of other critical functions will be privileged.

For performance reasons, most instructions are directly accessible by all privilege domains. Certain privileged instructions may be virtualized such that abstract data types are exported at the kernel interface. For example, the kernel may export an abstraction of the memory subsystem in the form of files, segment handles, or other objects. Finally, certain instructions will be reserved for kernel use alone, e.g. the *halt* instruction. Figure 2.5 illustrates these instruction differences.

2.5.2.2 Memory Protection, Process Address Space and Virtual Memory

To protect itself and protect processes from each other, the kernel must manage memory. Exclusive access to the memory management instructions ensures that the kernel can allocate memory for its own use and for the processes it creates. Depending upon the processor, this may involve management of segments, page tables, or both. Obviously, the address space accessible to the kernel is that of the entire processor, whereas that of the non-kernel applications is limited. When memory is accessed, the hardware checks the privilege level associated with the memory with that of the executing entity. For the memory access to be valid, the address must be within the address space of the process and must be accessible by the privilege level making the access. A kernel handler can return an exception if the address is not part of the valid address space.

Virtual memory adds another level of complexity to address space management. Virtual memory allows processes to have the appearance of an address space larger than the physical memory resources available in hardware. The virtual memory of each process is divided into small, equal-sized pages. Secondary storage, called swap space, is used to maintain the pages while the process is executing, and pages are swapped into and out of primary memory as needed. A process may only access pages that have been allocated to it. To maximize performance, this virtual-to-physical address space mapping is managed using combinations of hardware and software support. Detailed descriptions of virtual memory can be found in many articles, texts and manuals [29, 43, 47, 95].

2.5.2.3 Object Reuse Mechanisms

One way for adversaries to obtain information is through data scavenging. Although rummaging through the garbage to find papers that might contain sensitive information is a classic form of scavenging in the real world, digital data scavenging is an attractive corollary. Thus, no matter what policy is enforced, it is necessary to ensure that resources that may be reallocated to different processes are purged of information associated with their previous usage.

Objects encompass all information containers in a system, and the general term for this aspect of secure system implementation is *object reuse*. Because objects are pervasive elements in systems and are often shared, many techniques for ensuring their reusability have been developed [76]. Examples of memories that must be purged between use by different processes include primary memory, caches, secondary storage, buffers used by I/O devices, and various registers: essentially anything that could contain residual information from its use by a different process.

Ensuring that the objects to be purged are identified and managed correctly requires a systematic methodology, such as that discussed by Wichers [116]. Consideration must be given to how information objects are implemented in terms of combinations of initialization, allocation, deallocation and protection of the resource pool from which objects are created. For example, purging can be systematically performed before each new allocation of a resource, or after each deallocation. To determine the completeness of the object reuse implementation, a careful study of the system should be conducted. It should be noted that object reuse analysis differs from covert channel analysis because objects exported by and directly accessible via the system interface are intended as containers for information that must be shared or confined according to the security policy, whereas the system metadata used to establish covert channels are not intended to be accessible information containers.

2.5.2.4 Controlled Entry Points

If applications could invoke kernel functions arbitrarily, then the resource management services provided by the kernel would be obviated. Chaos would ensue: process isolation could not be guaranteed, and the kernel's internal resources could not be protected from manipulation by applications. Any hope of security policy enforcement would evaporate. To encapsulate the kernel and ensure that the use of the function is allowed and that only intended kernel functions are invoked by non-kernel entities, the kernel must provide a mechanism so that all calls to it can be controlled.

Hardware support is needed to accomplish this task. In the simplest case, a special instruction is invoked by the application layer that causes control to transfer to a special location in the kernel. There, the kernel will examine a predefined location for the parameter list. In addition to the usual parameters, an identifier for the desired function will be provided. The kernel should validate the parameters to ensure that pointers and address ranges are associated with the domain of the application and not that of the kernel. The kernel may then transfer control to the function that will process the call.

The mechanisms just described are sufficient for systems that have only two privilege domains, but a problem arises if the system has multiple privilege domains. If all the kernel can tell is that it has been invoked from a less privileged domain, then it is impossible to determine whether intermediate privilege domains are being protected from accidental or intentional abuse by even less privileged domains. An elegant solution to this problem of controlling inward calls is to use a gate mechanism [77], e.g. *call gates* [47]. These gates are placed at the boundary of each domain and control the transfer of execution from a less privileged domain to a more privileged one. They can be set up so that all calls into more privileged domains must cascade through a series of gates or so that a call can skip intermediate domains. This allows each domain to export its functions to selected lower privilege layers in the system. For example, the kernel may export certain kernel management functions only to the next most privileged layer and not to those with lesser privilege, thus providing the capability for the system builder to provide trusted code external to the kernel for management activities. In contrast, less trusted applications would be unable to invoke the kernel management functions. Intermediate domains may export abstract data types and the gates needed to invoke the type managers at the domain interface.

2.5.2.5 User Attribute Binding: The Trusted Path

Since the attributes bound to subjects acting on behalf of users are the basis for access control decisions, it is clear that a well-defined user identification and authentication policy is essential for secure systems. It was this observation that led Saltzer and Schroeder to include a trustworthy technique for identification and authentication in their list of fundamental requirements for protection [85].

Assuming that both the user and the identification and authentication mechanism are trustworthy, how can the user be sure that security-critical information being entered is not captured between the human interface and the I&A mechanism by a man-in-the-middle or some other malicious entity? An unforgeable connection that assures protected user communication with a trusted system mechanism is required. This is called a *trusted path*. Users invoke the trusted path using a *secure attention key*: a single key or special combination of keys or other input device intended solely for the purpose of establishing a trusted path. The secure attention key signal is received by the system's trusted mechanisms, and the I&A interface is displayed in a trustworthy manner to the user. Subsequent interaction is protected, and users have confidence that passwords and other critical authentication information is protected.

It is worth noting that a trusted path need not be restricted to the input of passwords: other critical information might require similar protection. The entry of bank account and credit card numbers, on-line confirmation of large financial transactions, electronic access to certain health records, or other high-value activities constitute examples where a trusted path provides enabling technology to organizations.

On a single platform, a trusted path requires that the interface presented to the user be constructed such that it depends only upon trustworthy mechanisms. This means that the use of large graphical user interface libraries of unknown or questionable provenance should not be within the dependency hierarchy of the trusted path mechanism. As a part of the system's overall security architecture, the trusted path must be as trustworthy as the components enforcing critical security policies.

A trusted path always refers to the interface between the user and the machine. Of course, in distributed systems, trusted communications between systems is also necessary. The term *trusted channel* is used to describe the protection of inter-system communications. In distributed systems, both trusted paths and trusted channels may require the use of cryptography. Care must be taken to ensure that the cryptographic functions and key management mechanisms are trustworthy.

User authentication to the system can be based upon any of three types of attributes: physical characteristics of the individual, something the individual knows, or something the individual has. Biometrics encompasses the use of physical characteristics for user authentication based upon physical characteristics. These include a number of modalities, common examples of which include fingerprints, voice recognition, retinal scans, iris scans, and facial recognition. Use of biometrics for access control requires that an initial biometric be enrolled. The template of the biometric of a claimant is compared to the enrolled template. Because of variations associated with biometric collection, an exact match of the two is extremely improbable, and statistical methods are used to determine whether to accept or reject a match. A number of significant research challenges, such as security, scalability, privacy, interoperability, and social aspects, need to be addressed to enable confident use of biometric technology for verifying identities [35, 49]. Passwords are something that the user knows and presents to the system to obtain access. Passwords do not suffer from the variations in collection described above, but they are vulnerable to guessing and brute-force attacks. A balance between the complexity of the password and user acceptability must be achieved. An example of an authentication mechanism involving something a user has is a physical token such as an ID card that may be presented to the system. Because tokens are susceptible to loss or theft, their use is often coupled with a second authentication mechanism. To provide higher confidence that only valid authentications occur and to limit the complexity of the individual mechanisms used, organizations often choose to combine authentication techniques. When two different methods are used to authenticate users, this is called *dual-factor authentication*, which naturally leads to *multi-factor authentication*, where the number of attributes is further expanded.

A wide range of authentication mechanisms is available, so system developers need to consider the effectiveness of the technical mechanisms against their usability, the context in which they will be employed, their cost, and their maintainability from both a physical and technical perspective.

If a user has one password for access to all systems, then the compromise of that single password renders the information being protected in all of the systems vulnerable. To mitigate this threat, it is recommended that users have different passwords for each different account. As the number of accounts proliferates, users must memorize an increasing number of passwords. A second problem may be encountered in enterprise systems where users may have to authenticate many times to access various services during a given session. In such cases, user frustration can be addressed through the implementation of *single sign-on* mechanisms. However, single sign-on has both advantages (e.g., fewer passwords to remember and enter) and disadvantages (e.g., greater damage if credentials are compromised).

2.5.2.6 Discretionary Policy Enforcement Mechanisms

Discretionary access controls are enforced by two kinds of mechanisms: access control lists and capabilities.

Access control lists (ACLs) itemize the access permissions of subjects to objects, such as files, directories, or devices. Each ACL entry consists of the name representing an entity, such as an individual user or group, and the rights accorded to that entity. Groups are convenient because access control lists for a large number of similar users (for example, all students enrolled in Psychology 101) can be simplified, thus reducing the possibility of administrator error. The largest possible group is everyone, which in many systems is termed *public*. Extremely simple ACLs are found in UNIX [5] and its descendant systems, such as Linux, where only a short set of *permission bits* are used to determine access to a file: owner, group, and public. For decades, systems intended for commercial use have had more sophisticated ACLs in which the permissions of particular individuals may be defined.

Not only can ACLs be used to permit access, but they may support the ability to deny access to particular subjects. Consider a file that provides answers to the exam to students following an exam. If Andy was out of town and must take the exam this week, the professor can explicitly deny Andy access to the answers until after he has completed the test. Thus the ACL may contain read permission for the group consisting of all members of the class and an additional entry that denys Andy access. The astute reader may have noticed that Andy as a member of the group *class* has been given access to the answers, so rules must be established regarding the precedence of the ACL permissions. In this case, the intent of the instructor is met if the ACL entries associated with individuals take precedence over the permissions accorded groups. Issues to be considered when determining ACL precedence rules are discussed by Lunt [68].

The simplest permissions found in ACLs may be merely read, write, and execute. Thus various users and groups will have one or more of these access rights to the object. Because discretionary access controls are often implemented for sophisticated applications, other types of access permission may be created. For example, it may be useful to allow certain users only append access to a file, for example a log file. In this case, writes are restricted to the end of the file. To implement append access, the underlying protection system will use a combination of both read and write. Without the user's knowledge, the system will open the log file, set the write pointer to the end of the file, and then write the next log record to the file. It is possible that the system or application programming interface will allow neither read nor write access explicitly to the user.

If ACLs contain the permissions to the objects, how are the permissions to the ACLs managed? This question is important because the runtime interface that permits modification of ACLs is what distinguishes them as elements of a discretionary access control mechanism. It is possible to associate *control* access rights with ACLs. The users or groups with control access to the object may be designated and will determine who can grant or deny permission for other access rights. These control access rights may be highly granular; for example, a particular individual might be given the ability to control a particular access right, e.g. read, within the ACL. Furthermore, the concept of control can be extended upward one more level so that certain individuals have control-of-control access rights. This rich set of access rights allows organizations to tailor discretionary access controls to meet specific requirements.

When new objects are created, it is important to ensure that the initial value of each ACL reflects the intended security policy. In some systems a template may be used to associate a default ACL with each new object. Such defaults may be system-wide or may be determined with higher granularity, for example, in the case of files, on a per-directory basis. Lunt [68] provided an analysis of discretionary control defaults, which can range from no access (i.e., minimized access) to complete access. In the context of least privilege, a default of limited or no access is a wise choice.

ACLs are attractive because all permissions associated with a particular object are localized. This allows policies to be managed easily. It is worth noting, however, that policy changes are not effective immediately. ACLs are used to check access permissions once the object is *opened*, prior to the actual read or write. The results of the access check are cached, and as long as the subject keeps the object *open*, the rights obtained at that *first* access are retained. Thus, revocation of access is not immediate and will only be effective the next time the user attempts to *first* access the object unless more sophisticated mechanisms are in place.

2.5.2.7 Capability Systems

Capabilities provide another way to implement discretionary access controls. In capability-based systems, which were first described by Dennis and van Horn [32], the list of access rights to objects are associated with the subjects, rather than the objects. In addition to defining access rights, capabilities provide a way to name objects, thus providing the basis for capability-based addressing [36]. For example, when a user logs onto the system, an initial set of capabilities is bound to the subject executing on the user's behalf. As execution progresses, subjects may accumulate additional capabilities. When a subject attempts to access an object, the RVM checks the access rights in the capability, and permission is granted if the requested access is included in those rights. Thus, once a subject possesses a capability for a particular object, that object may be accessed with the rights specified in the capability; all the subject needs to do is present the capability. A detailed discussion of capability systems is provided by Levy [65]. A notable implementation of a highly granular capability mechanism in an operating system was found in the CAP system [117].

Because a capability-based system distributes the access rights to each object among the subjects and since the rights may be stored as initialization data for each subject, revocation presents challenges. In addition, if subjects are able to copy and store capabilities, the revocation problem is further exacerbated. Also, there is no central location that can be inspected to determine which subjects have potential access to a particular object. Instead, the capability list for each subject must be inspected. If one decided to revoke access to an object, potentially every capability list in the system would require inspection to ensure that the revocation was complete. Again, as with ACLs, revocation would not take effect if the subject were already actively accessing the object. An approach to solve the revocation problem was proposed by Redell [80]. Capabilities can be particularly troubling in systems where mandatory policies are to be enforced because, in typical capability systems, no distinction is made between the access right and the ability to grant that access right [10]. The extension of capability systems to support lattice-based security policies was explored by Karger and Herbert [51, 52].

Although capability systems can be implemented, they are notoriously complex, and their lack of a conceptually simple policy-enforcement mechanism caused this approach to be eclipsed in terms of high assurance approaches. However, capability systems continue to be of interest, e.g. [92, 118].

2.5.2.8 Mandatory Security Policy Enforcement Mechanisms

Separation of domains requires the isolation of subsystems as well as mechanisms to allow controlled sharing between these domains.

2.5.2.9 Types of Mandatory Mechanisms

A *security kernel* binds internal sensitivity labels to exported resources and mediates access by subjects to other resources according to a partial ordering of the labels defined in an internal policy module [88]. The label space may support confidentiality and integrity policies as well as non-hierarchical categories [69]. A security kernel usually provides a hardware-supported ring abstraction [91, 93] and can host trusted subjects [89]. The rings can separate processes within a privilege level. Thus, a subject is a process-ring pair. All high assurance security kernels to date have utilized segmented memory, which provides persistent hardware based process-local memory-protection attributes [33, 38, 89, 90] as opposed to dynamic, global, hardware attributes based on memory paging mechanisms.

The security kernel mediates external communication via network devices that are each dedicated to a given sensitivity level, or via multilevel devices, in which a sensitivity label is bound to each network protocol data entity (e.g., datagram). Security kernels generally support full resource and resource-allocation configurability during runtime.

A *separation kernel* [83], which is sometimes referred to as a partitioning kernel [67], maps its set of exported resources onto partitions:

$$resource\_map : resource \rightarrow partition$$

Multiple *subject* resources and *object* resources may be mapped to a given partition, but a partition is an abstraction and is not itself a subject. Resources in a given partition are treated equivalently with respect to the inter-partition flow policy, and subjects in one partition can be allowed to access resources in another partition. Separation kernels enforce the separation of partitions and allow subjects in those partitions to cause flows, each of which, when projected to partition space (per the *resource_map* function), comprises a flow between partitions (which may be between different or identical partitions). The allowed inter-partition flows can be modeled as a *partition flow matrix* whose entries indicate the mode of the flow, similar to that of Fig. 2.2, discussed earlier

$$partition\_flow : partition \times partition \rightarrow mode$$

The mode indicates the direction of the flow, so that

$$partition\_flow(P_1, P_2) = W$$

means that subjects in P_1 are allowed to write to any resource in P_2. The assignment of resources to partitions and the access control or *flow* rules are passed to the separation kernel in the form of configuration data that the kernel interprets during system initialization. Since configuration data correctness is critical for the enforcement of the intended security policy, a configuration tool is often described for the construction of flow rules. Although not part of the kernel itself, this tool can help the security administrator or system integrator to organize and visualize complex data. This helps to ensure that user inputs reflect the intended policy.

Least privilege separation kernels (LPSKs) provide two important enforcement benefits beyond basic partitioning kernels. First, LPSKs increase the granularity of privileges accorded subjects as described in the Separation Kernel Protection Profile (SKPP) [46]. Second, unlike a partitioning kernel, an LPSK extends reference monitor features so that it is the locus of control for all inter-partition flows. In addition to the *resource_map* and *partition_flow* functions of a partitioning kernel, an LPSK supports the principle of least privilege in a manner than can be represented as a *subject-resource* flow matrix,

$$subj\_res\_flow : subject \times resource \rightarrow mode$$

It is possible to allow the subject-resource flow matrix to override the rules of the partition flow matrix [46]; however, a more restrictive interpretation, where a given flow is allowed by the LPSK only if both matrices allow it, is more intuitive and ultimately more likely to be correctly configured in system implementations [63]:

$$\begin{aligned} &allow\_flow(subject, resource, mode) \\ &\quad \rightarrow mode \in subj\_res\_flow(subject, resource) \quad \& \\ &\quad mode \in partition\_flow(subject.partition,\ resource.partition) \end{aligned}$$

The SKPP requires that

1. each secure configuration include an identification of a *base* partial ordering of flows between partitions to identify the strict MLS policy, and
2. subjects allowed to cause flows between partitions in addition to those base flows are treated as trusted subjects.

Figure 2.6 shows how the granularity of an MLS security policy can be refined through the application of the two policies. The baseline partial ordering appears in (a) and illustrates the partial ordering of the partitions: information may flow, as shown by the heavy arrows, from P1 to P2 and from P2 to P3, where the subjects within a partition are the entities that cause the flow. Least privilege is illustrated in (b). In this context, only certain subjects may cause flows, designated by lightweight arrows, to and from particular resources. For example, $S_{2,2}$ can only read from both O_{21} and O_{22}. A trusted subject, perhaps a specialized tool that downgrades only certain information, is shown in (c). It is permitted to cause a flow from O_{32} to O_{12}. As is the case for all trusted subjects, it is trusted to honor the intent of the system security policy and is thus shaded and has a dashed arrow to indicate a flow in opposition to those articulated in the base policy. Depending on the policy, an explicit partition rule allowing flow from P3 to P1 may be required for the S_{32} to O_{12} flow to be allowed.

A review of the relative merits of these three approaches is provided by Levin et al. [64].

2.5.2.10 Audit Mechanisms

A record of security-relevant events can be provided by an audit mechanism. If it includes dynamic rule-checking, the audit mechanism may provide alerts of impending security violations. Policies must be established to determine what should be audited. For example, audit might include: only accesses to a particular object; all activity on the system; the activities of subjects at a particular sensitivity level;

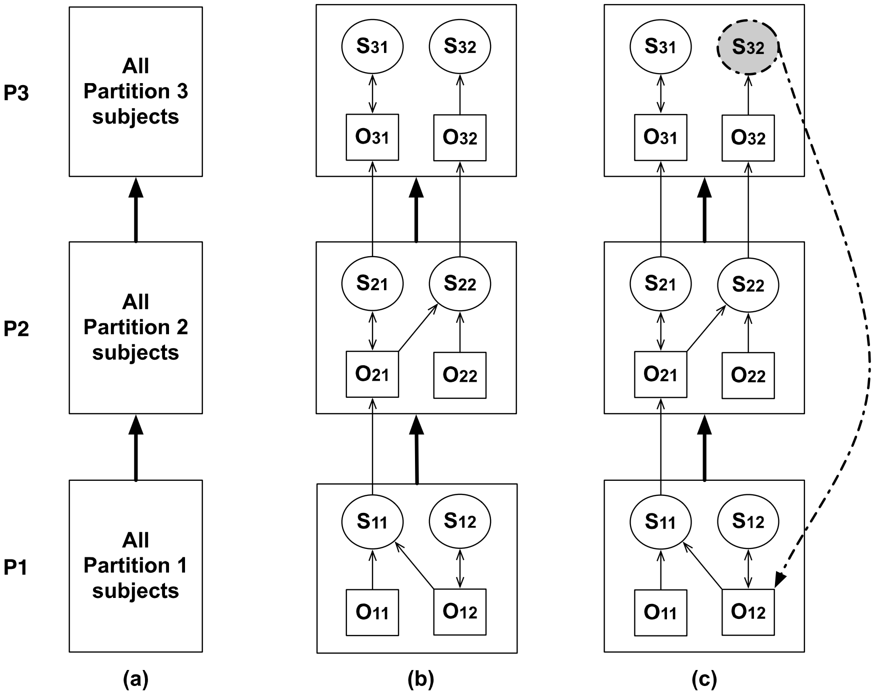


Fig. 2.6 SKPP policies. The partition_ flow policy is shown in (**a**), the more granular subject_resource policy in (**b**), and (**c**) illustrates a trusted subject

the use of selected system calls; etc. Because the security administrator and other trusted individuals engage in security-critical activities, an audit of their activities should be maintained. Also, good audit reduction tools are needed, otherwise voluminous audit records are not likely to be particularly useful.

Intrusion detection systems (IDS) constitute a dynamic form of auditing. Suggested by Anderson in 1980 [19], the next work on intrusion detection systems, published in 1987, provides a general IDS model [31]. Since that time, a wide variety of systems have been developed for network intrusion detection, e.g. Snort [96] and Bro [79], as well as host-based intrusion detection, e.g. Tripwire [55]. In the Snort and Bro systems, network traffic is closely monitored for patterns that would indicate the prelude to or initial steps of an intrusion, whereas the latter systems introspectively observe the activities on a single platform in an attempt to catch malicious activity prior to the completion of an attack. A fundamental limitation associated with intrusion detection systems is that they detect only what they are encoded to look for. Thus, however artful the encoding, the adversary can find a way around it. Security administrators are often presented with a choice between the reduction of false positives and the reduction of false negatives. IDSs can be thought of in terms of the following dichotomies: host-based vs. network based;

after-the-fact vs. real-time vs. predictive; and modeling misbehavior/detecting similarities vs. modeling good behavior/detecting deviations.

2.5.3 Composition of Trusted Components

To reduce costs in the construction of large systems, it is desirable to use existing commercial components as much as possible. In the parlance of FPGAs, this translates to the reuse of exiting IP. Composing secure embedded systems from multiple components presents several challenges.

2.5.3.1 Composition Problems

A classic example of problems introduced by composition is illustrated by the *cascade problem* [71]. The problem can be described as follows. Consider an MLS system where labels are linearly ordered by a comparison operator ($\geq$), and two labels are *adjoining* if there is not a label between them in the ordering. If a component enforces the security policy sufficiently to keep separate the information in two adjoining sensitivity levels, S_i and S_j, but no more, the component is said to have a level of trustworthiness of T_{two}. Let there be two T_{two} components, C_1 and C_2. Suppose that the organizational security policy requires the separation of three adjoining sensitivity levels: S_1, S_2 and S_3. The trustworthiness required for this separation is T_{three}. If C_1 separates S_1 and S_2 and C_2 separates S_2 and S_3, the architecture can be considered sufficiently trustworthy. However, if the components C_1 and C_2 are subsequently connected at the S_2 level as shown in Fig. 2.7, their combination forms a system—a virtual component—that spans three levels and yet has only T_{two} trustworthiness. Trustworthiness is not additive, so two serially linked T_{two} components are insufficient for a network policy that requires the separation of three sensitivity levels with a T_{three} level of trustworthiness.

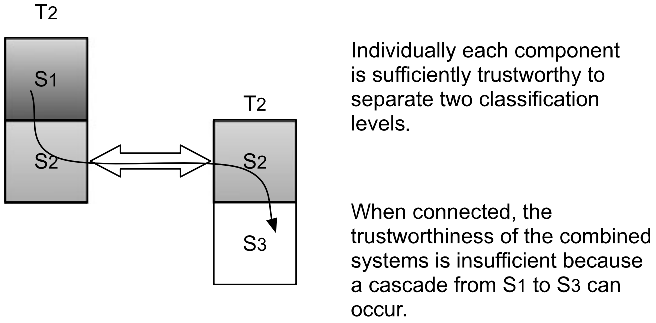

T2
S1
S2
T2
S2
S3
Individually each component is sufficiently trustworthy to separate two classification levels.
When connected, the trustworthiness of the combined systems is insufficient because a cascade from S1 to S3 can occur.

Fig. 2.7 The cascade problem. Separately the components are sufficiently trustworthy, yet when combined, their level of trustworthiness is insufficient

Analysis of the cascade problem shows that the algorithmic identification of a cascade within a network involves a time complexity of $O(an^3)$ and space complexity of $O(an^2)$, where a is the number of security levels and n is the number of nodes in the network [45]. Furthermore, the cost of calculating a correction (i.e., a policy-preserving reorganization of the network) is NP-Complete [45]. While reorganizing a network may only be a one-time cost, the conclusion to be drawn from this analysis is that it is better to avoid cascades in the first place.

An approach to avoiding the ad hoc nature of the composition problem is to provide a framework of rules under which pre-analyzed conjunctions of components may safely occur. For example, under TCB subsets [94], the system security policy is decomposed into a set of monitors, each of which enforces a subset of the overall policy. For example, one monitor might enforce the mandatory confidentiality policy, another the mandatory integrity policy, and yet another the discretionary confidentiality policy. A subject's access to objects is granted only when access is permitted by all three of the monitors. If the system can be subdivided such that separate components contain the monitors, then through appropriate engineering of a strict set of design and interface requirements, it may be possible to construct an architecture of these subset components that will generally satisfy the overall system policy. The goal, here, is to make it possible to construct the TCB subsets such that they may be evaluated independently, yet their composition results in enforcement of the larger system policy [75]. The result has been called a *partitioned TCB*.

Design Tip: *Composition*. Trustworthiness is not additive and may in certain circumstances be degenerative. Two components that are individually trustworthy are not necessarily trustworthy when put together. The TCB subset abstraction involves decomposing the security policy into a set of enforcement mechanisms, each of which enforces a subset of the overall policy. All subpolicies must be in agreement for access to be granted. However, the general result of *evaluation by parts* is still a hard problem because unintended behavior can *seep out of the box*.

2.6 Assurance of Policy Enforcement

Software and configurable hardware have many similarities as one examines assurance and lifecycle management practices employed for each class of technologies over a product's lifecycle.

An FPGA contains a set of logic elements that perform a specific function, and the programmable nature of an FPGA requires a means to specify the logic that defines the FPGA's behavior. Just as in software, which is defined in terms of a program expressed in a programming language, logic elements of the FPGA are typically expressed in a hardware description language. Furthermore, common FPGA logic elements may be expressed in libraries that are combined to produce progressively more complex functions. In this regard, an FPGA image may be considered a persistent and statically loaded program. Such a program can and should be subject to all the same analysis and assurance practices that are routinely applied to software.

Not unlike the offensive line on a football team, assurance is an often overlooked, but vital, element of maintaining a product through its entire lifecycle. Just as the line consistently performs the complex and unglamorous dirty work that allows the quarterback, running backs and receivers to be lauded for advancing the team down the field, the consistent, rigorous and successful application of sound assurance practices that result in a successful development effort is rarely recognized. Conversely, just as the offensive line often receives attention only when the quarterback gets sacked, assurance and configuration management practices typically receive the most scrutiny when flaws are uncovered.

But those knowledgeable about the sport know that the offense's success starts with the ability of the line to consistently open holes in the defense and provide the pass protection that allows the team to advance. The same is true of the assurance and configuration management processes that must be applied throughout the lifecycle of a robust and sound product.

2.6.1 Life Cycle Support

Life cycle management is an indispensable set of development and maintenance disciplines that helps define the assurance of a product, and thus the core competency of the manufacturer. Well-defined and efficiently managed life cycle models are the cornerstone to achieving design security for large and complex software projects. The actual life cycle processes vary among different organizations and depend on the product type (hardware versus software) and desired level of protection (high assurance versus low assurance). Nevertheless, these processes should apply the cradle-to-grave security principles during the entire life cycle of a product, viz. requirements engineering, design, development, manufacturing, testing, distribution, remediation and end-of-life disposal [108]. Although it is not a traditional focus of life cycle management and is often neglected, requirements engineering is an important aspect of security. Both functional and assurance (non-functional) security requirements must be correctly defined to avoid providing the wrong functionality or protecting the right functionality wrongly.

2.6.1.1 Assessment Criteria

The Common Criteria (CC), an internationally-recognized security evaluation framework, emphasizes the fundamental aspect of requirements engineering and prescribes a security requirements derivation methodology that is centered on a thorough analysis of both real and perceived threats to be mitigated by the end product [23]. Life cycle support plays an important role in the CC paradigm as evidenced by the large number of requirements devoted to life cycle modeling, configuration management (CM), secure delivery, developmental security, and flaw remediation [25]. Life cycle security issues related to programmable integrated circuits (e.g., FPGAs) are further addressed in a CC supplementary document [22] which provides guidance on how to apply the base CC evaluation methodology to hardware IC products that must be evaluated under the CC. In the US, all national security systems are mandated to be evaluated in accordance with the CC or NIST Federal Information Processing Standard (FIPS) validation program by the overarching National Security Telecommunications and Information Systems Security Policy No. 11 [20]. These systems often include programmable circuitry.

In the case of commercially available hardware cryptographic modules to be used in sensitive but unclassified environments, FIPS Publication 140-2 [104] is presently the official evaluation criteria which levies similar life cycle (albeit somewhat mislabeled as design assurance) requirements, i.e., configuration management, secure delivery and installation, developmental evidence, and operational guidance. FIPS 140-2 defines four hierarchical levels of security and explicitly refers to the CC for security requirements levied on trusted software used in the target crypto modules. This tie to the CC has been removed in the current draft FIPS 140-3 [107] which has been in a public review phase since July 2007. In this draft, design assurance was renamed to life cycle assurance, which includes additional requirements on the use of an automated CM system, vendor testing, and more rigorous development processes, e.g., the use of a high-level HDL for custom ICs starting at Security Level 2 [107].

2.6.1.2 Use of Trustworthy Tools

The draft FIPS 140-3 also requires that if software is included in the crypto module then information about the compilers, configuration settings, and methods used to generate the executable code must be provided, even at the lowest Security Level 1. This relates to the vexing *trustworthy tools* problem, i.e., how users can ascertain the correctness of the tools used to create executable code or to fabricate hardware circuits. For FPGAs, the problem is exacerbated due to the complexity of the tools used to design, manufacture, assemble, test, and distribute FPGA products. These tools are typically made by different vendors (both foreign and domestic), and there are no standardized metrics or criteria to assess the integrity of their implementation. In theory, formally verifying every tool would provide a high level of confidence that the end product is not subverted by the tools, but in practice, doing so would be prohibitively expensive. These challenges and other issues related to the trustworthiness of integrated circuits (both ASIC and FPGA) have been investigated and documented in the Defense Science Board study on High-Performance Microchip Supply [113]. This report had prompted DARPA to issue the Trust in Integrated Circuits research solicitation in 2007 [114], which focuses on "developing technologies that offer rigorous validation of IC hardware and its design regardless of where the design or manufacturing processes take place." The article entitled *The Hunt for the Kill Switch* [1] highlights the Trojan Horse attack and summarizes myriad vulnerabilities that are inherent in today's highly sophisticated hardware.

For secure software development, a *best practices* approach includes the following steps: (1) carefully selecting the tools based on common empirical analyses of quality factors such as provenance, maturity, stability and wide use, (2) performing a thorough black-box testing and security analysis of the tools' functional interfaces, and (3) maintaining the tools under strict configuration control. This process, if implemented properly, can help mitigate the threats of malicious subversion and accidental misuse. The testing and analysis results provide evidence to support the assertion that the selected tools do not introduce malicious functionality. Chapter 8 discusses as future work the application of this idea to the FPGA design, using a similar process to pick and manage the tools used in the different stages of the FPGA design flow (e.g., logic synthesis, place & route, etc.).

2.6.1.3 Applying Security Principles to Life Cycle Process

An effective life cycle methodology should incorporate the following high assurance software security principles as part of a defense-in-depth strategy:

- Audit
- Least privilege
- Separation of duties

Audit is a discipline of continuous inspection and assessment for accountability purposes. To detect security violations and deter penetration attempts, an audit framework should include both automated technical measures and manual actions. When applied to life cycle management, audit can ensure that all design and manufacturing activities conform to the life cycle control policies and procedures which can subsequently help offset the impacts of security breaches committed by malicious insiders. Deterrence is an effective risk management mechanism, and employing an audit policy that requires both random and periodic audit actions in all phases of a system's life cycle can also discourage potential adversaries from launching attacks. Configuration management (discussed below) is one form of auditing. It addresses the control of developmental and operational configuration changes that could affect the assurance disposition of a system.

Adherence to security principles such as least privilege and separation of duties that were suggested by Saltzer and Schroeder in their seminal work on protection of information [85] also affords additional protection and damage control. They defined least privilege as a design restriction that can limit the damages caused by both programmatic and operational errors, and separation of privileges (i.e., duties) as a protection mechanism that can reduce the risk of being compromised by colluding and maligned entities. The FPGA design and manufacturing flow is a complex series of different activities involving many actors and interdependencies, and verifying that the implementation of various components (e.g., netlist) has not deviated from the intended design (e.g., HDL design files) is a daunting task. While the use of cryptographic techniques can protect some parts of this flow [102], procedural safeguards based on the principles of least privilege and separation (i.e., isolating critical steps, enforcing distinct roles, and restricting privileges to the task at hand) can help strengthen the assurance posture of the end product.

2.6.2 Configuration Management

Technology is constantly evolving, and changes are unavoidable. Configuration management (CM) is a well-established practice to control changes that should be assimilated early in the life cycle for both software and hardware products. CM retrofitting (viz. adding CM as an after-thought) is costly and can severely impact a product's integrity since maintaining a complete and unmodified change history for traceability purposes is the bedrock of CM. CM can be viewed as an active defense mechanism that, when implemented properly, can help to mitigate inherent security risks associated with evolutionary changes.

Most developers are aware of CM, mostly as a versioning control mechanism, but they often choose to either ignore or marginalize the importance of CM for fear of being burdened by the CM controls and procedures. It is exactly those rigorous safeguards that, if properly practiced, could enable the establishment of the initial baselines of hardware and software components and the subsequent change control of those components. Change control plays a critical role in CM as its main objective is to prevent unauthorized modifications (including accidental errors) to the baseline configuration items. For high assurance software development, a thorough security analysis of the proposed changes (e.g., modifications of existing components and additions of new components) to assess the security impact on other parts of the system must be performed and reviewed prior to the approval of the change request. A system of checks and balances must be employed to deter collusion, e.g., clear separation of CM and the development environment, and to ensure the validity of the security analysis, e.g., to ensure that the analysis is performed by a trained security analyst and that the review is done by a Change Control Board. For FPGA development, the same CM objectives and requirements apply, especially for complex FPGA implementations that contain a large amount of code developed and maintained by a multitude of principals (core designer, system developer, manufacturer, etc.).

Configuration change control is important but is not enough. NIST has defined a set of CM requirements (i.e., security controls) that addresses a wide range of concerns ranging from establishing CM policy and procedures to maintaining a current inventory of the components used in a system [106]. Different combinations of these requirements are levied on information systems based on the system's potential impacts on an organization in the event of a security compromise. FIPS Publication 199 defines three levels of potential impacts—low, moderate, and high—based on the severity of the effect on the operations and assets of the organization, i.e., limited for low impact, serious for moderate impact, and severe or catastrophic for high impact [105]. NIST Special Publication 800-53 recommends the following eight CM requirement categories for moderate-impact and high-impact systems [106]:

- Configuration Management Policy and Procedures
- Baseline Configuration
- Configuration Change Control
- Monitoring Configuration Changes
- Access Restrictions for Change
- Configuration Settings
- Least Functionality
- Information System Component Inventory

These requirements cover all life cycle phases, i.e., planning, development and deployment, and they apply to all components of a system, including FPGAs. In other words, besides its effect on the developmental assurance of individual products, CM also contributes to a system's mission assurance if its use is included in the system's security strategy and plans. The security posture of a system is partially based on a set of approved configuration settings that, if changed without proper analysis and traceability, would invalidate the system's accreditation, i.e., license to operate. This is also true for FPGA-based embedded systems. Changes to a bitstream file in the field may have detrimental effects on both performance and security of a system; thus, CM policy and processes should be established and enforced to minimize the risks associated with bitstream reconfiguration.

2.6.3 Independent Assessment

Accountability and transparency are central elements in building and sustaining trust. Juvenal's poignant observation about trust[1] has been used over the years to emphasize the need for having external oversight to provide greater accountability in governance. It is easy to draw a parallel between this need for transparency and the need for security evaluation in secure product development since the security posture of a product could be strengthened if the product underwent an independent security assessment.

To be credible, the security evaluation of a product should be performed by an objective third party, preferably a government-sanctioned organization. This is because impartiality and independence are essential to guard against bias and collusion, respectively. In general, user confidence will increase if a vendor could demonstrate that their product passed the scrutiny of official organizations with legal oversight responsibility. For example, doctors and patients in the US would feel safer if the prescribed medicines for a life-threatening condition were approved by the Food and Drug Administration. Similarly, security-minded IT users in the US would be more

[1] "Quis custodiet ipsos custodies?" ("Who guards the guardians?")—Juvenal, Satires VI.347.

inclined to use security products that have been validated by official evaluation authorities such as the National Information Assurance Partnership (NIAP) Common Criteria Evaluation and Validation Scheme (CCEVS) and NIST.

CCEVS oversees and validates the evaluation of security products by CCEVS-approved commercial testing laboratories that are accredited by NIST [12]. The CC testing labs evaluate security products in accordance with the CC and NIAP-recognized Protection Profiles [13]. The CC is an international standard, and there are different evaluation schemes in other countries that provide the same CC evaluation oversight as CCEVS. In the US, the evaluation of cryptographic modules is performed by NIST, not CCEVS. NIST oversees the Cryptographic Module Validation Program (jointly with the Communications Security Establishment of the Government of Canada) which validates cryptographic modules in accordance with FIPS cryptographic standards, e.g., FIPS 140-2 [103].

Product evaluation authorities such as CCEVS and NIST only assess the trustworthiness of individual products (e.g., operating system, firewall, web server), not the trustworthiness of the end systems that use evaluated products. From a system acquisition viewpoint, independent security evaluation of individual products is a critical part of the technical due diligence which, when properly exercised, can help mitigate risks throughout the system's life cycle. However, a system that is composed of different evaluated products is not necessarily secure since the interactions among security functions provided by the evaluated products may result in new vulnerabilities. Product evaluation is performed based on security assumptions (e.g., physical and personnel security) and threats of specific operating environments for which a product is intended to be used. When integrated into an end system with a different threat model, the evaluated protection mechanisms may be inadequate to mitigate the threats manifested at the system level.

An independent critical examination of the integrated protection mechanisms at different dimensions of implementation (e.g., hardware, operating system, application software) could help identify adverse emergent behaviors prior to deploying the composed system for operational use. In the federal government, the process that federal agencies use for security and risk assessment before authorizing a system for operation is known as certification and accreditation (C&A). When there is a change in the functionality of an authorized system or its operational environment, subsequent C&A activities might ensue, depending on the organizational C&A policy, to determine and mitigate risks resulting from the change.

Although the complexity of the FPGA design in a product is typically hidden (encapsulated) in higher level functional components (e.g., processor cores and device controllers), it is important to not overlook the malleability of FPGAs in the security assessment of the overall system. The use of FPGAs should be inspected with the same depth and rigor that are used to assess the security of critical software in a product. Dynamic reconfiguration is an inherent benefit of using FPGAs, but it is also a double-edged sword. When FPGA-based products are used in a mission-critical system, it would be prudent to include, as part the system's C&A process, system-level architectural and design analyses to look for unintended side effects caused by poor, incorrect, or unanticipated use of FPGA-based components.

2.6.4 Dynamic Program Analysis

Dynamic program analysis generally refers to the testing and analysis of a program under execution. A target program is subjected to a specifically constructed set of input data, and instrumentation is used to examine and validate the program behavior. Input data may be constructed and instrumentation applied to:

- test for functional behavior
- test for performance
- test for timing constraints
- test for resource usage

Functional testing may be considered a common form of dynamic program analysis in which a program is subjected to a set of input data designed to exercise every interface of a program and validate all outputs in terms of effects, errors and exceptions. Functional testing is often conducted in conjunction with code coverage analysis to ensure that all program code gets exercised by the input data. The input data set is driven by code coverage and specifically designed to stimulate specific responses from the program.

In many environments, thorough testing of functional interfaces may be considered sufficient because properties such as performance, timing constraints or resource usage are not particularly demanding or may not be specified at all. Towards the other end of the spectrum are embedded, real-time systems in which resource and/or timing constraints may be absolute, severe and critical to the proper operation of the program. It is often difficult to know a priori how a program will behave in terms of performance or resource usage. In these environments, dynamic program analysis is applied to ensure the program behaves properly in response to a range of real world and pathological conditions.

2.6.4.1 Testing

Testing occurs in various contexts and at different phases during the development and certification of a software product. Unit and integration testing is typically conducted during development by the developers themselves. The rigor applied by developer-implemented testing can vary widely across organizations.

Once an overall software product is developed, it is subjected to a system test, typically conducted by a distinct Quality Assurance group. The test requirements are derived from a specification that completely describes the interface to the product. Testing procedures at this level are generally recognized to include:

- Thoroughly and completely exercising all interfaces to the program.
- Validating behavior under all externally visible states and conditions.
- Testing for both successful and unsuccessful conditions, including generation of all errors and exceptions.

If a product is subject to certification, then an evaluator may conduct additional testing. The examination and testing of a product can vary widely depending on the nature of the certification [26, 109]. Such testing may be comprised of simply running a defined test suite against the product to verify compliant functional behavior. However, often the certification also seeks to assess the compliance of a product at not just a single point it time, but over the lifetime of the product. Under this type of requirement, the certification process must examine not only the product itself but also all the software development practices applied to develop and maintain the product. Often the evaluator cannot examine, much less repeat, all the testing conducted by the developer. Instead, the evaluator examines testing processes, test records, documentation, and other materials that demonstrate software lifecycle assurance.

While testing is recognized as a vital process in software development, it is often compartmentalized to discrete phases within the development lifecycle, typically after a software unit or even a complete program has been coded.

The quality of a software product can be greatly improved by addressing testing requirements throughout the development lifecycle, not only by conducting testing at appropriate points within the development process, but also by actively considering testing impacts during the design and implementation of a program. A *design for test* strategy includes both application of design principles that promote simple and appropriately constructed interfaces and application of coding techniques that facilitate testing.

Application of design principles, including developing a sound abstraction and creating an appropriately modular architecture, tend to promote more intuitive and simpler programs that are thus easier to test. Interestingly, while principles of abstraction and modularization might not be easily grasped, more rote examination of an implementation can yield equally valuable feedback. For example, an interface regarded as *hard to test* or having *too many test cases* suggests an interface that might be unnecessarily complex given an understanding of a particular functional requirement.

Application of techniques that facilitate testing can enable development of extensive unit and integration test suites that may be used to support initial development and regression testing. Unit tests are typically conducted using a test driver that can exercise the unit under test by not only invoking the interfaces exposed by the software unit but also by applying code practices that allow one to selectively expose and control the internal state of the software unit.

Similar to the traditional software development process, the FPGA development process typically involves several iterative steps in which the output of each progressive step of development is fed back to verify the functional behavior of the circuit. To support this iterative model, the FPGA development tools have evolved to support sophisticated hardware description languages and testing techniques that can be used to construct test fixtures to exercise implementation logic and interfaces of an HDL circuit description at each stage of the FPGA design flow.

2.6.5 Trusted Distribution

It is important that the delivery of trusted products is protected against counterfeiting and subversion during transit from the vendor site to the user site. The user must have the following guarantees on the received product:

- It is of the correct version as specified by the vendor. If the product had been evaluated, the distributed version must also match the evaluated configuration.
- It comes from the vendor, not from a fraudulent source, and
- It arrives unmodified.

This type of assurance requirements is characterized in evaluation criteria as *trusted distribution* [110] and *secure delivery* [25], respectively.[2] The need for these requirements stems from the fact that any unauthorized changes to a product's security mechanisms during its life cycle could have an adverse effect on the system's ability to enforce its security policy. While configuration management provides protection against subversion during the development phase, trusted distribution addresses threats of subversion and forgery during the distribution phase.

In TCSEC, trusted distribution requirements are only levied on Class A1 products since it was considered too costly for lower assurance classes to provide assurance measures for ensuring secure delivery [111]. Specifically, the TCSEC requires the vendor to implement a distribution system that can ensure the integrity of the delivered product and to provide procedures for users to validate that the received version is the same as the distribution master's version [110]. These requirements apply to both the initial delivery and subsequent updates of a product. Accompanying the TCSEC is a series of technical guidelines whose purpose is to clarify the TCSEC requirements and to provide implementation guidance. *A Guide to Understanding Trusted Distribution in Trusted Systems* [111] is one such document. This guide explains why trusted distribution is an important life cycle assurance measure and provides insights on different approaches to implementing an effective trusted distribution mechanism.

The Common Criteria, on the other hand, imposes trusted distribution requirements starting at Evaluation Assurance Level 2 (EAL2), the second lowest level of a seven-level assurance scale. In previous CC versions (Version 2.3 and older), trusted distribution requirements were grouped into one family (ADO_DEL) and expressed in terms of *delivery procedures* (EAL2 and EAL3), *detection of modification* (EAL4 through EAL6), and *prevention of modification* (EAL7) [21]. These categories are linearly hierarchical, i.e., detection of modification requires delivery procedures, and prevention of modification requires both detection of modification and delivery procedures. These requirements are similar to the TCSEC requirements in that they focus on the use of the vendor's master copy and address both procedures and technical measures employed at both ends of the delivery channel.

In the current CC Version 3.1, trusted distribution requirements are expressed in terms of *delivery procedures* and *preparative procedures* [25]. The former re-

[2] For the purpose of this discussion, the two terms are considered to be equivalent.

quires the vendor to document and use the delivery procedures for distributing the product. The latter requires the vendor to provide acceptance procedures for users at the user site. The CC assurance requirements underwent a major rewrite after Version 2.3, and trusted distribution requirements are now defined as two separate families (ALC_DEL and AGD_PRE), making it harder for CC novices to follow. Anti-subversion and anti-counterfeit safeguards that were explicitly specified in the previous trusted distribution requirements have been made into application notes which are not normative in the CC paradigm.

Although the trusted distribution requirements in the TCSEC and CC are different in scope and form, they all have the same objective of protecting the product against post-development subversion and theft. Similar threats also exist in FPGAs, and just as in high assurance software, stringent delivery mechanisms should be employed to mitigate these threats at different development stages of an FPGA-based system.

2.6.6 Trusted Recovery

A secure system (implementing a state-machine model) must ensure that each state transition after an initial *secure state* results in another secure state [8]. Although the definition of secure state depends on a system's security policy model, in general a secure state can be viewed as a system state in which the system data is consistent and uncorrupted, and the system can correctly enforce the security policy represented by the its security model [46].

When a system detects that it is no longer in a secure state, it must attempt to self-recover to a secure state without further protection compromise while recovery is in progress. The concept of recovering in the presence of abnormity while ensuring continuity of protection is known as *trusted recovery*. The TCSEC and CC further characterize these exceptions as either *failure* or *discontinuity of operations*. A failure can be an error condition in the system's *security functionality*[3] that causes the system to behave incorrectly (e.g., inconsistent values in system data structures caused by transient hardware failure) or a media failure (e.g., a disk crash). A discontinuity of operation, on the other hand, is an error caused by inappropriate human actions, e.g., inappropriate shutdown of a system [24, 112].

While a system is running (as opposed to halted), it can be in one of two modes: operational or maintenance. Trusted recovery mechanisms must be supported in both modes [46]. These mechanisms must be able to determine whether the current system state is secure or not and to initiate mode-specific recovery actions to repair the system if the system is not in a secure state. Certain error conditions can be recovered by automated mechanisms (e.g., remapping of a bad disk sector) while

[3] The term *security functionality* is based on the term *TOE Security Functionality (TSF)* which is defined in the CC as a set consisting of all hardware, software, and firmware of the TOE that must be relied upon for the correct enforcement of the security functional requirements [23].

others require manual recovery actions (e.g., system crash due to an unexpected error). Recovery methods also vary depending on the operational environment of the system. For example, the recovery method used in an embedded real-time system would require more complex processing than the method used in a traditional computer due to resource constraints of the embedded system (e.g., processor overhead and response time) [62].

Although self-testing is a system integrity requirement, it is also relevant to trusted recovery. Its use during system initialization and normal operation can detect abnormal conditions that require recovery. Moreover, self-tests can be invoked by an automated recovery mechanism as part of the recovery process or performed by an administrator to verify that the system is indeed in a secure state after completing a recovery action.

Regarding life cycle assurance, recovery mechanisms must meet the same development assurance requirements levied on other security-relevant functions, since they are parts of the system's security functionality. Their design and implementation must be critically reviewed to ensure that architectural properties such as self-protection, least privilege, modularity and minimization are upheld, and that no Trojan horses or trap doors exist in the code. Furthermore, security testing and vulnerability analysis must be performed to determine potential vulnerabilities that could be exploited to bypass security enforcement during recovery [26].

Recovering from certain failures may require complex administrative actions. Since administrative users typically have more privileges than regular users, the principle of least privilege should be applied to the assignment of recovery privileges such that only authorized administrative users (e.g., the security administrator, not the system operator) can perform recovery functions. The operational guidance documentation must describe all types of failure conditions, recovery procedures, and tools, and for each type of failure, specific guidance on how best to recover from the failure. It is paramount that user documentation on recovery is complete and correct; otherwise, misuse of recovery functions may affect the system's ability to fail securely, resulting in compromised protection.

2.6.7 Static Analysis of Program Specifications

As opposed to testing the properties of a program[4] in execution, *static analysis* provides assurance of the properties of a program based on an objective examination of some specification of the program.

Programs can be specified at different levels of abstraction, for example, user manuals, design specs, source code, and executable code are different abstractions of a program, and different forms of static analysis examine different levels of abstraction. Often, the term *static analysis* is used to refer specifically to the analysis of source code [17], but in this context, it is used more broadly.

[4] Where *program* could be a module, component, monolithic system, or distributed system.

2.6.7.1 Code Reviews and Bug Checking

A basic form of static analysis occurs during *code reviews*, in which program authors along with peers, designers, managers, and customers, etc., look at source code. A code review can focus on properties such as elegance, faithfulness to a higher level specification, and coding errors like buffer overflows. Automated static analysis tools [6, 15, 16, 28] can perform some of these analyses of specifications, as long as the properties in question can be defined in terms of an *effective procedure* (e.g., a rule) understandable by the tool. For example, while a tool might be able to search for a well-defined buffer overflow, many concepts of program elegance are subjective and beyond the scope of current tools. Some source code properties that are statically checked by automated tools include: correct syntax and format, memory leakage, improper stack or general-memory access, memory leaks, overuse of privilege, buffer overflow, unused or duplicate functions, unused variables, uninitialized variables, lack of encapsulation/data-hiding, and time-of-check to time-of-use errors.

Automated code reviews and bug checking, like automated testing, may not provide adequate assurance that a property is completely or correctly implemented in a given program (the theoretical difficulty of writing a program to understand other programs is related to the *halting problem* [11]). The use of formal methods, discussed next, can lead to greater assurance of program security.

2.6.7.2 Formal Methods

Human languages lend themselves to ambiguity and lack of precision, whereas mathematics provides a basis for clear and precise description and reasoning about those descriptions. *Formal methods* is not, itself, a formally-defined or standardized term of art in computer science, but in general, it refers to the application of mathematics in various aspects of the software and hardware system development process. In particular, the use of formal methods to verify *security* properties is required for *high assurance* or *high robustness* [14] product ratings.

Some formal methods are:

- general mathematical models of
  - computation and processing [44]
  - security [8, 9]
- specification languages [84, 97, 98] with precise semantics that can express:
  - system behavior
  - security properties
  - refinement relationships between specifications
  - theorems of conformance to properties
  - theorems of conformance to more abstract specifications
- executable security languages with precise semantics:
  - in which security properties such as correct MLS flow can be described with first order language constructs

  - successful compilation of a program guarantees that it conforms to the properties that it includes [30, 115]
- automated systems for manipulating the logic of formal specifications, such as:
  - automatic or interactive theorem provers [54, 78, 84]
  - model checkers, model executers, and SAT solvers [48]
  - tools that automatically generate theorems based on properties in a formal specification [34]
- information flow analysis tools [34]

2.6.7.3 Refinement and Preservation of Properties

Formal verification of a secure system includes formalization of key specifications, at different levels of abstraction, as well as a series of correspondence demonstrations showing that each specification preserves the security properties of the next most abstract level—resulting in a transitive argument that the implementation preserves the security policy (see Fig. 2.8). The more the elements of this chain are formalized, the more formal the resulting argument. The particularization of a given specification to one that is more concrete (i.e., less abstract) is called *refinement*; whereas the translation of a concrete specification to one that is more general is called *abstraction*.

The prevalent criteria for high assurance verification [14, 110] have required several items in common: a formal specification of both the security policy model and the top level functional specification (an interface specification that includes the inputs, outputs, processing, and internal effects of each interface); a proof that the formal model is consistent with its own security properties; and a proof that the formal specification preserves the properties of the model. Formal methods may also be used in the analysis of covert channels and in the demonstration that the source code is consistent with the formal top level specification.

The usual expectation is that the natural language security policy and the security policy model are simple enough that their consistency can be ensured through inspection, with a high degree of confidence. The security policy model is often a

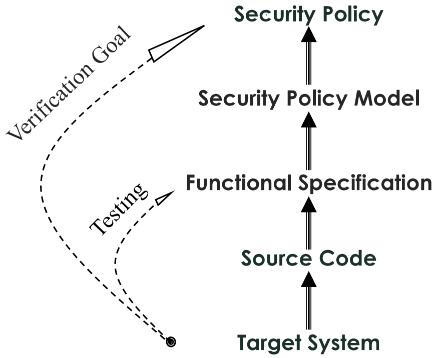


Fig. 2.8 Formal verification chain of evidence

refinement of the security policy, where an organizational-level policy that is "independent of the use of a computer" [110] is interpreted in the computer technology domain—i.e., the security policy model helps to transition between the security policy and the formal specifications. On the other hand, verified translation of source code to machine code (i.e., *trusted compilers*) and the automatic translation of formal functional specifications to source or machine code [97] are topics of current research.

In what has been called the *refinement paradox*, [70, 82] it has been shown that the refinement of an *information flow* model [39] does not, in general, preserve the security properties of the model—e.g., the addition of detail (viz., in the formal specification) may introduce flows not included in the abstract formal model. In this case, covert channel analysis can be performed to ensure that the information flow in the refined specification is correct. Similarly, if an access control model [8] is used, a covert channel analysis of the formal specification ensures that information flow that is extraneous to the model does not violate the security policy.

The correct correspondence of source code to the formal specification can be demonstrated through exhaustive enumeration of the source code, in which each element of code is mapped to its representation in the formal specification and is accompanied by a rationale as to why the semantics of the formal specification are preserved in the refinement. One of the goals of research to provide verified automatic translation of the formal specification to source code is to avoid the arduous manual code-correspondence task as well as to reduce the error rates associated with manual coding.

References

1. S. Adee, The hunt for the kill switch. IEEE Spectrum **45**(5), 34–39 (2008)
2. P. Ammann, R.S. Sandhu, The extended schematic protection model. J. Comput. Secur. **1**(3, 4), 335–385 (1992)
3. J.P. Anderson, Computer security technology planning study. Tech. Rep. ESD-TR-73-51, Air Force Electronic Systems Division, Hanscom AFB, Bedford, MA, 1972. Also available as vol. I, DITCAD-758206. Vol. II, DITCAD-772806
4. E.A. Anderson, C.E. Irvine, R.R. Schell, Subversion as a threat in information warfare. J. Inf. Warfare **3**(2), 52–65 (2004)
5. M.J. Bach, *The Design of the UNIX Operating System* (Prentice Hall, Inc., Englewood Cliffs, 1986)
6. T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S.K.R. Jamani, A. Ustuner, Thorough static analysis of device drivers. SIGOPS Oper. Syst. Rev. **40**(4), 73–85 (2006)
7. D.E. Bell, L. LaPadula, Secure computer system: unified exposition and multics interpretation. Tech. Rep. ESD-TR-75-306, MITRE Corp., Hanscom AFB, MA, 1975
8. D.E. Bell, L. LaPadula, Secure computer systems: mathematical foundations and model. Tech. Rep. M74-244, MITRE Corp., Bedford, MA, 1973
9. K.J. Biba, Integrity considerations for secure computer systems. Tech. Rep. ESD-TR-76-372, MITRE Corp., 1977
10. E.W. Bobert, On the inability of an unmodified capability machine to enforce the *-property, in *Proceedings DoD/NBS Computer Security Conference*, September 1984, pp. 291–293

11. G. Boolos, R. Jeffrey, *Computability and Logic* (Cambridge University Press, Cambridge, 1974)
12. CCEVS, Publication #4: guidance to CCEVS approved Common Criteria testing laboratories, version 2.0. National Information Assurance Partnership Common Criteria Evaluation and Validation Scheme, September 2008
13. CCEVS, Publication #1: organization, management and concept of operations, version 2.0. National Information Assurance Partnership Common Criteria Evaluation and Validation Scheme, September 2008
14. CCMB, Common Criteria for information technology security evaluation, revision 3.1, revision 1, no. CCMB-2006-09-001. Common Criteria Maintenance Board, September 2006
15. B.E. Chelf, S.A. Hallem, A.C. Chou, Systems and methods for performing static analysis on source code. US Patent 7,340,726, Coverity, Inc., 2008
16. H. Chen, D. Wagner, MOPS: an infrastructure for examining security properties of software, in *Proc. 9th ACM Conf. Computer and Communications Security (CCS 02)*
17. B. Chess, G. McGraw, Static analysis for security. IEEE Secur. Priv. **2**, 76–79 (2004)
18. S. Christy, R.A. Martin, Vulnerability type distributions in CVE. http://cve.mitre.org/docs/vuln-trends/index.html, May 2007
19. J.P.A. Co, Computer security threat monitoring and surveillance. Tech. Rep., James P. Anderson Co., Fort Washington, PA 19034, February 1980
20. Committee on National Security Systems, NSTISSP no. 11, revised fact sheet. National Information Assurance Acquisition Policy, July 2003
21. Common Criteria Maintenance Board, Common Criteria for information technology security evaluation, part 3: security assurance components, version 2.3, CCMB-2005-08-003. Common Criteria Maintenance Board, August 2005
22. Common Criteria Development Board, The application of CC to integrated circuits, version 2.0, revision 1, CCDB-2006-04-003. Supporting document, mandatory technical document. Common Criteria Development Board, April 2006
23. Common Criteria Maintenance Board, Common Criteria for information technology security evaluation, part 1: introduction and general model, version 3.1, revision 1, CCMB-2006-09-001. Common Criteria Maintenance Board, September 2006
24. Common Criteria Maintenance Board, Common Criteria for information technology security evaluation, part 2: security functional components, version 3.1, revision 2, CCMB-2007-09-002. Common Criteria Maintenance Board, September 2007
25. Common Criteria Maintenance Board, Common Criteria for information technology security evaluation, part 3: security assurance components, version 3.1, revision 2, CCMB-2007-09-003. Common Criteria Maintenance Board, September 2007
26. Common Criteria Maintenance Board, Common Criteria for information technology security evaluation, evaluation methodology, version 3.1, revision 2, CCMB-2007-09-004. Common Criteria Maintenance Board, September 2007
27. M.A. Cusumano, Who is liable for bugs and security flaws in software? Commun. ACM **47**, 25–27 (2004)
28. M. Das, S. Lerner, M. Seigle, ESP: path-sensitive program verification in polynomial time, in PLDI 02: Programming Language Design and Implementation, June 2002, pp. 57–68
29. P.J. Denning, Virtual memory. ACM Comput. Surv. **2**(3), 153–189 (1970)
30. D.E. Denning, A lattice model of secure information flow. Commun. ACM **19**(5), 236–243 (1976)
31. D.E. Denning, An intrusion-detection model. IEEE Trans. Softw. Eng. **13**, 222–232 (1987)
32. J.B. Dennis, E.C.V. Horn, Programming semantics for multiprogrammed computations. Commun. ACM **9**(3), 143–155 (1966)
33. DigitalNet Government Solutions, Security target version 1.7 for XTS-6.0.E, March 2004
34. P. Eggert, D. Cooper, S. Eckmann, J. Gingerich, S. Holtsberg, N. Kelem, R. Martin, FDM user guide. No. TM-8486/000/04, Reston, VA: Unisys Corporation, June 1992
35. European Commission, Biometrics at the frontiers: assessing the impact on society. Tech. Rep., European Commission Joint Research Center (DG JRC), Institute for Prospective Technological Studies, 2005

36. R. Fabry, Capability-based addressing. Commun. ACM **17**, 403–412 (1974)
37. R. Fitzgerald, trans. *Homer: The Odyssey* (Vintage, New York, 1961)
38. L.J. Fraim, Scomp: a solution to the multilevel security problem. Computer **16**, 26–34 (1983)
39. J. Goguen, J. Meseguer, Security policies and security models, in *Proc. of 1982 IEEE Symposium on Security and Privacy*, Oakland, CA (IEEE Comput. Soc., Los Alamitos, 1982), pp. 11–20
40. G.S. Graham, P.J. Denning, Protection—principles and practice, in *Proceedings of the Spring Joint Computer Conference*, May 1972, pp. 417–429
41. I. Hadzic, S. Udani, J. Smith. FPGA viruses, in *Proceedings of the Ninth International Workshop on Field-Programmable Logic and Applications (FPL'99)*, Glasgow, UK, August 1999
42. M. Harrison, W. Ruzzo, J. Ullman, Protection in operating systems. Commun. ACM **19**(8), 461–471 (1976)
43. J.L. Hennessy, D.A. Patterson, *Computer Architecture: A Quantitative Approach*, 4th edn. (Morgan Kaufmann, San Mateo, 2006)
44. C.A.R. Hoare, Communicating sequential processes. Commun. ACM **21**(8), 666–677 (1978)
45. J. Horton, R. Harland, E. Ashby, R.H. Cooper, W.F. Hyslop, B. Nickerson, W.M. Stewart, O. Ward, The cascade vulnerability problem, in *Proceedings IEEE Symposium on Research in Security and Privacy*, Oakland, CA, May 1993, pp. 110–116
46. IAD (Information Assurance Directorate), US Government protection profile for separation kernels in environments requiring high robustness. National Information Assurance Partnership, version 1.03 edn., 29 June 2007
47. Intel, Intel 64 and IA32 architectures software developer's manual, vol. 3A: system programming guide, part 1. Intel Corporation, Denver, CO, 253668-022us edn., November 2006
48. D. Jackson, *Software Abstractions: Logic, Language, and Analysis* (MIT Press, Cambridge, 2006)
49. A.K. Jain, S. Pankanti, S. Prabhakar, L. Hong, A. Ross, J.L. Wayman, Biometrics: a grand challenge, in *Proceedings of the 17th International Conference on Pattern Recognition*, August 2004, pp. 935–942
50. M.J. Kaminskas, *Risk Assessment/Risk Management*. Building Design for Homeland Security, vol. 5. FEMA, Risk Management Series ed. (2007). http://www.fema.gov/library/viewRecord.do?id=1939
51. P.A. Karger, Improving security performance for capability systems. Ph.D. thesis, University of Cambridge, Cambridge, England, 1988
52. P. Karger, A.J. Herbert, An augmented capability architecture to support lattice security and traceability of access, in *Proceedings 1984 IEEE Symposium on Security and Privacy*, Oakland, CA (IEEE Comput. Soc., Los Alamitos, 1984), pp. 2–12
53. P.A. Karger, R.R. Schell, Multics security evaluation: vulnerability analysis. Tech. Rep. ESD-TR-74-193, vol. II, HQ Electronic Systems Division, Air Force Systems Command, Hanscom Field, Bedford, MA 01731, June 1974
54. M. Kaufmann, J. Moore, An industrial strength theorem prover for a logic based on common Lisp. IEEE Trans. Softw. Eng. **23**(4), 203–213 (1997)
55. G.H. Kim, E.H. Spafford, The design and implementation of Tripwire: a file system integrity checker, in *Proceedings of the 2nd ACM Conference on Computing and Communications Security (CCS)*, Fairfax, VA, November 1994
56. P. Kocher, Timing attacks on implementations of Diffie-Hellman, RSA, DSS and other systems, in *Proceedings of the 16th Annual International Cryptology Conference (CRYPTO)*, Santa Barbara, CA, August 1996
57. M. Kurdziel, J. Fitton, Baseline requirements for government and military encryption algorithms, in *MILCOM*, vol. 2, Oct. 2002, pp. 1491–1497
58. L. Lack, Using the bootstrap concept to build an adaptable and compact subversion artifice. Master's thesis, Naval Postgraduate School, Monterey, CA, June 2003
59. B.W. Lampson, Protection, in *Proc. 5th Princeton Conf. on Information Sciences and Systems*, Princeton, NJ, 1971
60. B.W. Lampson, A note on the confinement problem. Commun. ACM **16**(10), 613–615 (1973)

61. C.E. Landwehr, Formal models for computer security. ACM Comput. Surv. **13**(3), 247–278 (1981)
62. K. Lee, L. Sha, Process resurrection: a fast recovery mechanism for real-time embedded systems, in *Proceedings of 11th IEEE Real Time and Embedded Technology and Applications Symposium 2005* (RTAS 2005), March 2005, pp. 292–301
63. T.E. Levin, C.E. Irvine, T.D. Nguyen, Least privilege in separation kernels, in *E-business and Telecommunication Networks; Third International Conference*, ed. by J. Filipe, M.S. Obaidat. ICETE 2006, Set'ubal, Portugal, 7–10 August 2006. Communications in Computer and Information Science, vol. 9 (Springer, Berlin, 2008)
64. T.E. Levin, C.E. Irvine, C. Weissman, T.D. Nguyen, Analysis of three multilevel security architectures, in *Proceedings 1st Computer Security Architecture Workshop*, Fairfax, VA, November 2007, pp. 37–46
65. H.M. Levy, *Capability-based Computer Systems* (Digital Press, Bedford, 1984)
66. S. Lipner, The trustworthy computing security development lifecycle, in *Proceedings 20th Annual Computer Security Applications Conference* (IEEE Comput. Soc., Los Alamitos, 2004), pp. 2–13
67. Lockheed-Martin/The Open Group, Protection Profile for PKS in environments requiring high robustness. Draft Version 1.3, submittal for NSA approval, 09 June 2003. http://www.csds.uidaho.edu/pp/PKPP1_3.pdf. Last accessed: 15 March 2009
68. T.F. Lunt, Access control policies: some unanswered questions. Comput. Secur. **8**, 43–54 (1989)
69. T.F. Lunt, P.G. Neumann, D.E. Denning, R.R. Schell, M. Heckman, W.R. Shockley, Secure distributed data views security policy and interpretation for DMBS for a Class A1 DBMS. Tech. Rep. RADC-TR-89-313, vol. I, Rome Air Development Center, Griffiss, Air Force Base, NY, December 1989
70. J. McLean, Security models and information flow, in *Proceedings of the IEEE Symposium on Security and Privacy* (IEEE Comput. Soc., Los Alamitos, 1990), pp. 180–189
71. J. Millen, The cascading problem for interconnected networks, in *Fourth Aerospace Computer Security Applications Conference*, 1988, pp. 269–273
72. J. Murray, An exfiltration subversion demonstration. Master's thesis, Naval Postgraduate School, Monterey, CA, June 2003
73. S. Myagmar, A. Lee, W. Yurcik, Threat modeling as a basis for security requirements, in *Proc. Symp. Requirements Engineering for Information Security (SREIS 05)*, 2005
74. P. Myers, Subversion: the neglected aspect of computer security. M.S. thesis, Naval Postgraduate School, Monterey, CA, 1980
75. National Computer Security Center, Trusted network interpretation of the trusted computer system evaluation criteria, NCSC-TG-005, July 1987
76. National Computer Security Center, A guide to understanding object reuse in trusted systems. Tech. Rep. NCSC TG-018, National Computer Security Center, Fort George G. Meade, MD, 1991
77. E.I. Organick, *The Multics System: An Examination of Its Structure* (MIT Press, Cambridge, 1972)
78. L.C. Paulson, *Isabelle: A Generic Theorem Prover*. LNCS, vol. 828 (Springer, Berlin, 1994)
79. V. Paxon, Bro: a system for detecting network intruders in real-time. Comput. Netw. **31**(23–24), 2435–2463 (1999)
80. D. Redell, R. Fabry, Selective Revocation of Capabilities, *International Workshop on Protection in Operating Systems*, IRIA, 1974
81. D. Rogers, A framework for dynamic subversion. Master's thesis, Naval Postgraduate School, Monterey, CA, June 2003
82. A. Roscoe, CSP and determinism in security modelling, in *Proceedings of the IEEE Symposium on Security and Privacy* (IEEE Comput. Soc., Los Alamitos, 1995), pp. 114–127
83. J. Rushby, Design and verification of secure systems. *ACM SIGOPS Operating Systems Review*, vol. 15, December 1981, p. 12
84. J. Rushby, S. Owre, N. Shankar, Subtypes for specifications: predicate subtyping in PVS. IEEE Trans. Softw. Eng. **24**(9), 709–720 (1998)

85. J.H. Saltzer, M.D. Schroeder, The protection of information in computer systems. Proc. IEEE **63**(9), 1278–1308 (1975)
86. R. Sandu, Analysis of acyclic attenuating systems for the SSR protection model, in *Proceedings of the 1985 IEEE Symposium on Security and Privacy*, April 1985, pp. 197–206
87. R.S. Sandhu, The schematic protection model: its definition and analysis for acyclic attenuating schemes. J. ACM **35**, 404–432 (1988)
88. R.R. Schell, P.J. Downey, G.J. Popek, Preliminary notes on the design of secure military computer systems. Tech. Rep. MCI-73-1, Electronic Systems Division, Air Force Systems Command, Hanscom AFB, Bedford, MA, 73
89. R. Schell, T.F. Tao, M. Heckman, Designing the GEMSOS security kernel for security and performance, in *Proceedings 8th DoD/NBS Computer Security Conference*, 1985, pp. 108–119
90. D.D. Schnackenberg, Development of a multilevel secure local area network, in *Proceedings of the 8th National Computer Security Conference*, October 1985, pp. 97–101
91. M.D. Schroeder, J.H. Saltzer, A hardware architecture for implementing protection rings. Commun. ACM **15**(3), 157–170 (1972)
92. J.S. Shapiro, J.M. Smith, D.J. Farber, EROS: a fast capability system, in *SOSP'99: Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles* (ACM, New York, 1999), pp. 170–185
93. L.J. Shirley, R.R. Schell, Mechanism sufficiency validation by assignment, in *Proceedings 1981 IEEE Symposium on Security and Privacy*, Oakland (IEEE Comput. Soc., Los Alamitos, 1981), pp. 26–32
94. W.R. Shockley, R.R. Schell, TCB subsets for incremental evaluation, in *Proceedings Third AIAA Conference on Computer Security*, December 1987, pp. 131–139
95. A. Silberschatz, P.B. Galvin, G. Gagne, *Operating System Concepts*, 7th edn. (Wiley, New York, 2005)
96. Snort.org, Snort. http://www.snort.org/, last referenced 22 March 2009
97. Specware 4.2 Manual, Kestrel Technology, http://www.specware.org/documentation/4.2/languagemanual/SpecwareLanguageManual.pdf, 3 November 2008
98. J.M. Spivey, *Understanding Z: A Specification Language and Its Formal Semantics* (Cambridge University Press, Cambridge, 1988)
99. D.F. Sterne, On the buzzword "security policy", in *Proceedings of the IEEE Symposium on Research on Security and Privacy*, Oakland, CA (IEEE Comput. Soc., Los Alamitos, 1991), pp. 219–230
100. The Easter Egg Archive, Excel Easter Egg—Excel 97 flight to credits. http://www.eeggs.com/items/718.html, last accessed 19 February 2009
101. K. Thompson, Reflections on trusting trust. Commun. ACM **27**(8), 761–763 (1984)
102. S. Trimberger, Trusted design in FPGAs, in *Proceedings of the 44th Design Automation Conference*, San Diego, CA, June 2007
103. US Department of Commerce and Communications Security Establishment of the Government of Canada, Implementation guidance for FIPS PUB 140-2 and the cryptographic module validation program, initial release: 28 March 2003, last update: 10 March 2009. National Institute of Standards and Technology, Gaithersburg, MD, March 2009
104. US Department of Commerce, Security requirements for cryptographic modules, Federal Information Processing Standards Publication 140-2. National Institute of Standards and Technology, Gaithersburg, MD, May 2001
105. US Department of Commerce, Standards for security categorization of federal information and information systems, Federal Information Processing Standards Publication 199. National Institute of Standards and Technology, Gaithersburg, MD, February 2004
106. US Department of Commerce, Recommended security controls for federal information systems, NIST Special Publication 800-53 Revision 2. National Institute of Standards and Technology, Gaithersburg, MD, December 2007
107. US Department of Commerce, Security requirements for cryptographic modules, Federal Information Processing Standards Publication 140-3 (Draft: 07-13-2007). National Institute of Standards and Technology, Gaithersburg, MD, July 2007

108. US Department of Commerce, Security considerations in the system development life cycle, NIST Special Publication 800-64 Revision 2. National Institute of Standards and Technology, Gaithersburg, MD, October 2008
109. US Department of Commerce, Derived test requirements for FIPS PUB 140-2, Security requirements for cryptographic modules, 24 March 2004, Draft, CMVP program staff (NIST, CSE and CMVP laboratories). National Institute of Standards and Technology, Gaithersburg, MD. http://csrc.nist.gov/groups/STM/cmvp/documents/fips140-2/fips1402DTR.pdf. Cited 7 April 2009
110. US Department of Defense, Trusted computer systems evaluation criteria (Orange Book) 5200.28-STD. National Computer Security Center, Fort Meade, MD, Dec. 1985
111. US Department of Defense, A guide to understanding trusted distribution in trusted systems, version 2, NCSC-TG-008. National Computer Security Center, Fort Meade, MD, December 1988
112. US Department of Defense, A guide to understanding trusted recovery in trusted systems, version 1, NCSC-TG-022. National Computer Security Center, Fort Meade, MD, December 1991
113. US Department of Defense, Defense Science Board task force on high performance microchip supply. Office of the Under Secretary of Defense For Acquisition, Technology, and Logistics, Washington, DC, February 2005
114. US Department of Defense, TRUST in integrated circuits, presolicitation notice, solicitation number: BAA07-24. Defense Advanced Research Project Agency, Microsystems Technology Office, Arlington, VA, March 2007. http://www.darpa.mil/mto/solicitations/baa07-24/index.html, cited 27 Mar 2009
115. D. Volpano, C. Irvine, Secure flow typing. Comput. Secur. **16**(2), 137–144 (1997)
116. D.R. Wichers, Conducting an object reuse study, in *Proceedings of the 13th National Computer Security Conference*, October 1990, pp. 738–747
117. M.V. Wilkes, R.M. Needham, The Cambridge model distributed system. ACM SIGOPS Oper. Syst. Rev. **14**(1), 21–29 (1980)
118. E. Witchel, J. Cates, K. Asanovic, Mondrian memory protection, in *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, San Jose, CA, October 2002
119. C. Zymaris, A comparison of the GPL and the Microsoft EULA. 2003. Cybersource. Retrieved 15 September 2008, from http://www.cybersource.com.au/cyber/about/comparing_the_gpl_to_eula.pdf

Chapter 3
Hardware Security Challenges

Abstract This chapter discusses the problem of malicious hardware, or *gateware*, on FPGAs. Categories of malicious hardware, the problem of foundry trust, and attacks facilitated by malicious inclusions are presented. This chapter also explains the problem of covert channels on FPGAs, with a formal definition of a covert channel in general and a description of the specific case of covert channels on FPGAs. Methods for detecting and mitigating these covert channels are also described.

3.1 Malicious Hardware

Chapter 2 described results from the world of high assurance software that can also be applied to FPGA designs. Beginning with software makes sense because of the vast body of work on computer security as it relates to software. Because the modern FPGA design process resembles software development in several aspects (hardware description languages vs. high-level programming languages, reuse of intellectual property, etc.), the term *gateware* has been coined to describe circuit designs that are loaded into the gates of an FPGA.

The history of malicious hardware can be traced to the Cold War when the Americans and the Russians spied on each other and continues to the present day. Russia intercepted typewriters destined for America and inserted keylogger functionality [23]. The Europeans may have added a *kill switch* to a processor that prevented Syrian radar from detecting an Israeli attack [5].

3.1.1 Categories of Malicious Hardware

A malicious function can be implemented in either software or hardware. Malicious hardware has many of the properties of malicious software, and lessons learned from the categorization of malicious software into taxonomies can be applied to malicious hardware as well. Countering malicious hardware is difficult because determining

T. Huffmire et al., *Handbook of FPGA Design Security*,
DOI 10.1007/978-90-481-9157-4_3, © Springer Science+Business Media B.V. 2010

the trustworthiness of an arbitrary computer program (or hardware module) is not decidable in the general case, since such analysis is equivalent to the halting problem, according to Rice's theorem [31]. Mitigation of malicious hardware requires secure design practices, including mandatory access control mechanisms, formal verification of secure systems, and configuration management.

A *backdoor* or *trapdoor* allows unauthorized users to access a system. They can be installed during system development or during the installation of system updates.

A *kill switch* is a type of subversive artifact that allows the attacker to disable the functionality of hardware or software [5]. Like a trap door, a kill switch can be installed during system development or during maintenance, but instead of enabling illicit access, it can cause denial-of-service (DoS). A kill switch can be installed as part of a bitstream, or it can be put onto the chip by third-party development tools or malicious insiders at an overseas foundry.

FPGA viruses have been demonstrated that can configure the FPGA to short-circuit, resulting in melting the device [11]. Viruses can spread via software (malicious software code with a malicious FPGA configuration payload) or hardware replication (much more difficult than software). Although FPGA viruses are not common *in the wild*, for sensitive applications, the possibility of even an unlikely event must be considered. The same can be said about successfully reverse engineering an FPGA bitstream: it is difficult but must be considered in certain circumstances.

A *Hardware Trojan* is a term of art in the hardware security community referring to malicious alterations and inclusions in ICs [38]. Wang, Tehranipoor, and Plusquellic have developed a framework for the classification of malicious alterations to ICs, a taxonomy that is useful for evaluating methods for their detection [38]. Wang et al. classify hardware Trojans based on their physical, activation, and action characteristics. Physical traits can be divided into type (realized by adding or deleting transistors or gates), size (number of added, deleted, or compromised chip components), distribution (location in the physical layout of the chip, either close together or dispersed), and structure. The taxonomy of Wang et al. uses four physical attributes, one activation attribute, and one action attribute, for a total of six attributes [38]. Hardware Trojans can either be activated externally or internally. Trojan action is classified into modification of the chip's function, modification of the chip's parametric properties (e.g., delay), and transmitting key information to an adversary. Detecting and mitigating hardware Trojans is an active area of research.

3.1.2 Foundry Trust

In the world of hardware trust, the Defense Advanced Research Projects Agency (DARPA) has created a program called TRUST in Integrated Circuits (TIC) to detect malicious inclusions in ASIC chips using both invasive (e.g., sand-and-scan) and non-invasive (e.g., X-ray) detection techniques [32]. The TRUST program is also concerned with the security of FPGAs and the protection of third-party intellectual property (IP). The US Department of Defense and the National Security Agency (NSA) have begun the Trusted Foundry Program to address concerns about the security of chips used in sensitive government systems, and a few foundries have received certification [9, 23].

TIC was established by government officials concerned with the problem that it is impractical to manufacture every chip used in critical systems in a trusted foundry. A fighter jet, for example, can have on the order of a hundred processors. Since the military has to save money, the use of commodity components cannot be avoided in the real world. The TRUST program is divided into several groups, or teams, and each group consists of a partnership between industry and academia [32]. The Government Support Teams consist of the Red Team, the Test Article Generation Team, and the Metrics Team. The Test Article Generation Team, led by the University of Southern California Information Sciences Institute (USC-ISI), uses MOSIS to fabricate the test chips in commercial foundries. The Red Team, led by MIT Lincoln Lab, identifies various types of malicious circuits and develops methods of inserting these circuits into some of the test chips. Three Performer Teams attempt to detect the malicious inclusions: Xradia's specialty is nondestructive X-ray techniques; Luna's specialty is antitamper features for FPGAs; and Raytheon's specialty is hardware and logic testing [5]. The Metrics Team, based at the Johns Hopkins University Applied Physics Laboratory, establishes metrics for judging the outcome of the experiments (i.e., to measure success). To stay in the program, the performer teams must meet strict detection rate and false positive rate goals. In addition to the problem of malicious IC insertion in military systems, the TRUST program is also concerned with ASIC and FPGA design flows [32], which may use third-party IP, which poses a thorny problem.

Trimberger discusses the trusted foundry problem as it relates to FPGAs [36]. Since the foundry has no idea which chip will be delivered to which customer, application-specific attacks are difficult to execute. Furthermore, since the FPGA is programmed after manufacturing, the foundry has no idea what design will be loaded onto the device. The fine granularity of the programmability of the FPGA makes it difficult for the foundry to gain knowledge about the design. For example, CPUs, which have less granularity of programmability than FPGAs, are an easier target for attackers because much is known about how CPUs execute programs and about how CPUs are structured. Attackers can target certain parts of the CPU, and only around one thousand gates are needed to give an attacker full control of the system [17]. FPGAs, on the other hand, are arrays of programmable logic, and a malicious person at the foundry cannot predict how the application will map to the FPGA, increasing the difficulty of an attack. However, there are parts of an FPGA that are dedicated to specific functionality, and these may be the target of subversion. For example, hard-wired CPUs, hard-wired SRAM and BRAM blocks, and standard functions for I/O, for loading bitstreams, or for crypto key storage.

Reconfigurability is useful to defenders who can move data and cores around the chip to make it even harder for attackers. Defenders can use redundancy to their advantage by forcing attackers to modify all redundant versions of the design in precisely the same way. Clearly, some of these ideas resemble security through obscurity, and the security of a system should not rest solely on its blueprints being secret. However, when used in conjunction with more formally sound methods, they can increase the cost of attacking the system.

> *Design Tip*: *Security Through Obscurity*. Don't base the security of a system solely on the secrecy of its blueprints. Assume that the enemy may be able to obtain the blueprints or reverse engineer your system. Cryptosystems require the secrecy of the *keys*, but the *ciphers* are published so that the worldwide crypto community can scrutinize them. Study the blueprints of your FPGA's security mechanisms. If they are trade secrets, consider alternative manufacturers if the security of the system depends too much on the confidentiality of the mechanisms.

3.1.3 Physical Attacks

The intentional insertion of malicious inclusions into circuits makes it possible to subsequently mount attacks on those circuits (vulnerabilities may also exist in hardware due to unintentional design flaws). In a physical attack, the attacker has physical control of the device. Attacks may be sorted into one of three categories: non-invasive, semi-invasive, and invasive, which describe the degree of physical intrusion to the target system. For example, if the attacker has physical control over a smart card terminal, a *non-invasive* side channel attack against a smart card in the terminal is possible. Such an attack may use simple power analysis, differential power analysis, or fault injection against the crypto circuitry to obtain crypto keys, since the smart card relies on the power supplied by the terminal. Power analysis attacks are also possible on FPGAs [34]. Side channel attacks that involve removing the packing but not physically altering or damaging the chip are considered to be *semi-invasive* attacks. Removing the packaging can make it easier to analyze the electromatic radiation emitted by the chip. An example of an *invasive* attack is the *sand-and-scan* attack in which the passivation layers of an integrated circuit are systematically removed and scanned by an electron microscope [7]. Chemicals, lasers, or focused ion beams can be used to remove the layers. Another example of an invasive physical attack is the use of chemical solvents to remove the packaging from a smart card and then using a probing station to probe the bus traffic. A very sophisticated physical attack involves the use of a focused ion beam workstation to chemically drill through the potting material surrounding a smart card and then laying down metal shunts to probe a processor without disturbing the tamper-resistant mesh surrounding it.

Design Tip: *Tamper Resistance, Bitstream Encryption, and Determined Adversaries*. Given sufficient resources, an adversary can overcome tamper resistance techniques. Amateurs learned how to defeat smart card tamper resistance mechanisms in order to watch satellite TV for free [6]. Your risk assessment should consider differential power analysis attacks against the bitstream detection mechanism. A thorough security evaluation of the bitstream decryption mechanisms of different vendors is useful information when selecting an FPGA platform.

The sand-and-scan attack is the quintessential example of a device-level or chip-level attack, in contrast to the relatively easy board-level attack, which probes the metal pins of a circuit board. Security architects often define a logical security perimeter or boundary around some or all of the components that fit on a chip, such as the processor and caches for the purpose of analysis and documentation. For example, a hardware design may encrypt data before it leaves the *secure* area of the chip and decrypt it when it enters the area; it also may perform integrity checks on the data when it enters the secure area [18, 22, 24, 25, 39].

Design Tip: *Security Perimeters*. Beware of establishing a logical security perimeter or boundary and then assuming that everything within that perimeter is secure. Assume that a determined adversary will be able to breach any such *Maginot line*, given sufficient resources.

Mitigating physical attacks is extremely difficult, and a threat model that takes into account the resources available to the adversary is essential. While some physical attacks can be carried out by amateurs with equipment from the hardware store, some attacks require highly specialized knowledge and equipment. For example, the focused ion beam station, used in the semiconductor industry for legitimate purposes, is also useful for some types of physical attacks and costs millions of dollars for the attacker to obtain. Working with nanometer feature sizes also increases the challenge for the adversary. Many techniques developed for CPUs and ASICs are also applicable to FPGAs, such as the tamper resistance mechanisms developed for the IBM 4758, which is surrounded by a tamper-sensitive wire mesh and epoxy *potting* material [33].

3.2 Covert Channel Definition

A first order requirement for the enforcement of most security policies is the isolation of active entities like processes and cores, such that specific communication between them can be allowed and controlled in an orderly manner. Isolation is supported in part by virtualization of various shared physical resources, which provides each process or core with a distinct virtual resource that cannot be interfered with by other entities.

3.2.1 The Process Abstraction

For general-purpose processors, the *process* abstraction simplifies software development by unifying various processing elements, such as registers, threads, program counters, code segments, and program data segments, as an entity for reasoning about behavior within the computer. Processes can be comprised of multiple threads and rings, each of which has distinct security characteristics; therefore, the subject abstraction (e.g., a process/ring pair) is often used to more precisely reason about active entities with respect to security. Similarly, a dedicated single-purpose core on a multi-core processor could be considered to be a subject.

3.2.2 Equivalence Classes

Another simplifying technique for secure system design is to group like entities into a domain or *equivalence class*, so that there are fewer things to keep track of. Multi-domain security policies partition system subjects and objects into equivalence classes by binding each subject to a sensitivity label. Virtual machines and multi-core processors can be configured to partition computer resources into equivalence classes, such that all of the processes with similar security attributes are assigned to the same VM or core, or multiple cores are assigned to the same equivalence class with respect to the policy. This allows the designer of an access control mechanism to focus on the actions of subjects that cross equivalence class boundaries.

Design Tip: *Equivalence Classes*. The equivalence class abstraction can help make your design more secure and efficient. Grouping similar subjects and objects into domains reduces policy complexity, decreases enforcement mechanism overhead, and simplifies security analysis.

3.2.3 Formal Definition

A covert channel is a means to transfer information between subjects in a manner that is not intended for information transfer and is not allowed by the system security policy [15]. Incomplete virtualization of a resource, in which contention for the resource is visible to different subjects—for example, whether or not the disk is full, or the processor is occupied—provides a *point of interference* around which a covert channel can be constructed. The manner in which the covert channel receiver detects the interference differentiates covert *storage* channels (the receiver is provided different system call status messages) from covert *timing* channels (the receiver views changes to the relative timing of events).

3.2.4 Synchronization

To stream data via a covert channel usually requires precise synchronization on both the sending (*high* in terms of mandatory confidentiality policies) and receiving (*low*) side to repeat the interference event. Covert channels that do not require high side synchronization—that is, the low side essentially eavesdrops on the high side's behavior without cooperation of a malicious or Trojan horse high-side subject—are called *side channels*. Since synchronization of the unauthorized transmission of data is not available in this scenario, side channels are usually specific to a fixed-sized datum such as a cryptographic key.

3.2.5 Shared Resources

To enhance performance, modern processors attempt full utilization of their resources (such as instruction cache, data cache, the floating point arithmetic unit, and the branch prediction unit) [1–4, 14] by sharing them at the micro-architectural level between threads, cores, and processors. It is up to the OS to virtualize these resources to prevent interference between subjects of different equivalence classes. For example, even if the L1 and L2 caches are logically isolated, the L3 cache may provide a point of interference if the virtualization is incomplete. In a cache *side-channel*, one subject's use of a given cache line increases the response time of the next subject that uses it, even if the two subjects use the cache line for different memory addresses; this can disclose the data used for encryption and eventually the encryption key. Alternatively, if use of the floating point unit is mutually exclusive, one subject can experience a delay if another subject is using it.

3.2.6 Requirements

In general, a covert channel (with many variations in the literature) is based on several factors:

1. The victim and receiver are in different equivalence classes (as above).

2. Use of a shared resource by a victim (the *sender*) and the attacker (the *receiver*)—e.g., a set of cache lines.
3. A means to initiate and synchronize sender and receiver actions—often the attacker can initiate sender actions at will, as in the case of an enterprise's end-to-end VPN. With a side channel, the sender could be completely unaware and provide no synchronization.
4. A means to detect if another subject is using or has used the resource—a difference in response time when the attacker reads memory via that cache line.
5. Interpretation of the information in a side channel may require knowledge of the relationship between the key and the victim's use of a memory structure in a cryptographic function such as an *S-box*.

3.2.7 Bypass

In addition to covert channels and side channels, there are also *direct channels*, also known as *overt channels* or *bypass*. A direct channel simply means that security mechanisms do not exist (or are not applied) to prevent communication between two entities (subjects) in the system [21]. Direct channels often have high bandwidth. For example, in a system that has no memory protection mechanism, two cores can communicate with each other via external memory by writing to and reading from a specific shared memory region. Another example of a direct channel is a system that *does* have a memory protection mechanism, but two cores can communicate via external memory by bypassing the memory protection mechanism. Another example of a direct channel is a system with multiple cores communicating over a shared bus, if that bus has no arbitration mechanism to prevent illegal communication.

3.3 Existing Approaches to Limiting Covert and Side Channel Attacks

The design of micro-architectural shared resources, without corresponding ISA primitives to secure that sharing (e.g., the task and segment management features of the Intel iAPX86 processor come to mind as examples with such primitives), puts operating system designers, who must design mechanisms to manage hardware in a secure manner, in a difficult situation. Previous software approaches have been to either ignore the problem or to employ a heavyweight strategy that has an onerous performance impact.

3.3.1 Shared Resource Matrix Methodology

Kemmerer [15, 16, 21] has devised a shared resource matrix method of identifying covert storage and timing channels in computer systems. All shared resources that can be referenced or modified by a subject are enumerated, and each resource is carefully examined to detect whether it can be used to transfer information from one subject to another covertly. The rows of the shared resource matrix represent all shared resources and the attributes of those resources that are visible to subjects. There is a column for each operation. The analysis determines the entries of the matrix by identifying the operations that reference or modify the attributes. The matrix reveals when the sender and receiver have access to the same attribute of a shared object, the sender can modify it, and the receiver can reference it. As discussed above, a covert channel also requires a mechanism for initiating the processes of both sender and receiver and sequencing their accesses.

3.3.2 Cache Interference

Cache interference on uni-processors has often been dealt with by *normalizing* the cache (e.g., evicting all cache lines) between execution of different security domains (i.e., during process context switches). It is very time consuming to replenish the cache from off-chip memory, and various techniques have been developed to avoid doing so unnecessarily [12]. However, this approach is ineffective for processors supporting concurrent execution—chip multi-processors (CMPs), as well as single-core computers with simultaneous multithreading (SMT), and symmetric multiprocessor (SMP) systems with cache coherency mechanisms—as access to the cache is interleaved at a far more granular level than process switching. In these systems, micro-architectural interference has been a significant challenge since its efficient solution appears to lie beyond the capability of the usual operating system virtualization techniques. Recent research has responded with mixed results. For example, the cache can be physically or logically partitioned per policy equivalence class (if virtual cache support is available in hardware) [30, 37], which requires modification to the processor in the case of physical partitioning and reduces the effective cache size in both physical and logical partitioning. Various forms of cache disablement are possible, ranging from turning it off, to turning it off for certain cores or processes [28], all of which result in response time penalties. Lowering the bandwidth of the cache channel [29, 30, 35] may leave the design open to future exploitations as this approach does not eliminate the covert channel's essential point of interference. Application-level mitigations against features of specific cryptographic algorithms suffer from the same problem [1].

3.3.3 FPGA Masking Schemes

Although there is much prior work on timing, power, and electromagnetic side channels [10, 19, 20], some of which is applicable to FPGAs, techniques for making ASICs resistant to side channel attacks do not necessary work on FPGAs because of the differences in the way that FPGAs and ASICs implement logic gates and circuits. Yu and Schaumont have developed a technique for creating FPGA designs that are resistant to side channels [40]. Their technique involves building a complementary, symmetrical circuit that *masks* the power consumption of its dual. Chen and Schaumont also present a hardware masking scheme that uses algorithmic masking with multi-bit masks to make it more challenging for attackers to successfully estimate the secret internal random mask bit in circuit-level masking [8].

3.4 Detecting and Mitigating Covert Channels on FPGAs

Today's FPGAs are very powerful: undergraduate students can use commodity FPGA development boards and design tools to build embedded systems-on-chip (SoCs). A typical design has multiple Intellectual Property (IP) cores, and each core performs a specific function. As in the case of software, hardware may have security flaws, and a vulnerable core could leak information.

3.4.1 Design Flows

Covert channels, side channels, and direct channels can be introduced during the design phase by subverting the design tools (similar to subverting a compiler) or by subverting the IP cores (similar to subverting a shared library). As in the software world, FPGA design involves many stages, and each stage is very complex. There are many possible design flows involving different combinations of individual tools, and one tool is often just one stage of a larger design flow. As is true in software, code reuse is essential to managing design cost, and IP cores are often reused in many designs. Designing every component of a system from scratch is not feasible, given the scale and complexity of today's design projects.

3.4.2 Spatial Isolation

The solutions proposed in this book to the problem of covert channels on FPGAs exploit the reconfigurable nature of FPGAs, which allows the designer to build in reconfigurable hardware whatever security mechanisms are needed. These solutions also exploit the spatial nature of FPGAs, which allows the designer to place cores in specific regions of the FPGA. Chapter 6 describes a technique that spatially isolates cores during the layout portion of the design phase in order to prevent cores from interfering with each other. With this *moats and drawbridges* technique, the moats provide isolation, and the drawbridges provide controlled sharing in a well-defined manner. Chapter 6 also describes the specific example of a covert timing channel in multi-core systems that communicate via a shared bus and proposes a solution to this problem that incorporates a TDMA bus arbitration mechanism.

3.4.3 Memory Protection

Chapter 5 describes a memory protection technique for preventing cores from interfering with each other. Specifically, a *reference validation mechanism* in reconfigurable hardware enforces a security policy that specifies the legal sharing of memory among cores. The reference monitor denies all illegal memory access requests by cores, which prevents the off-chip memory from being used as a means of communication between cores. A precisely defined language for expressing memory access policies is used in conjunction with a design flow for compiling a policy directly to a reconfigurable reference monitor circuit that enforces the policy.

3.5 Policy State as a Covert Storage Channel

As Chap. 5 will explain formally, a *reference monitor* decides to either grant or deny a particular memory access request according to the policy it enforces. There are two kinds of policies: *stateless* policies that only have one state, and *stateful* policies that have two or more, with transitions between them. With certain stateful policies, the internal state of the reference monitor could be used as a shared resource in a covert storage channel attack. A *high* sender core can send information to a *low* receiver core by changing the internal state of the reference monitor in a way that the receiver core can observe. The ability to reference and modify the internal state of the reference monitor facilitates illegal communication [13].

3.5.1 Stateful Policies

A stateful policy can be represented as a directed graph, which contains transitions (directed edges) between multiple states (nodes). The graphs of some stateful policies have *cycles*, and if certain conditions are met, a cycle may represent a *possible* covert channel. The most conservative course of action is to revise the policy in order to eliminate the cycle. However, if such revisions are not feasible, one technique for coping with the covert channel is to monitor its rate of usage and if the rate exceeds a specified threshold, over time, take corrective action. The rate is measured with a counter which is incremented every time the cycle completes, indicating that one bit of information can flow. Several options for corrective action are discussed later in this section.

3.5.2 Covert Channel Mechanism

Every covert channel has a sender and a receiver, and in an MLS system the sender has a higher confidentiality label than the receiver. On an FPGA supporting an MLS

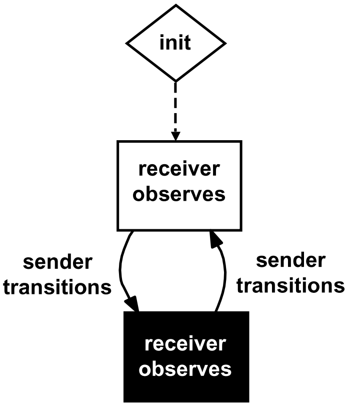


Fig. 3.1 A stateful policy can be represented as a directed graph, which contains transitions (directed edges) between multiple states (nodes). This graph has a cycle that indicates a possible covert channel. The sender can alternate between the two states at will, and the receiver can observe the difference between the two states

policy where individual cores are allocated to different security levels, the sender and receiver are cores. The sender alternates the internal state of the reference monitor by making access requests that cause transitions between the states of the policy. The receiver can observe this change if the receiver is permitted to perform an access in one state but is not permitted in the other state. In other words, the two states differ with respect to the operations allowed to the receiver. Since the transmission alphabet has two characters (white and black states in Fig. 3.1) this provides one bit of information to flow from the sender to the receiver. Construction of a *stream* of information based on this interference event requires there to be a cycle in the graph involving two or more states. Every time the cycle completes, one bit of information can flow, and the state is reset to enable the next cycle. Ideally, the cycle only has two states, and the sender can cause both transitions of the cycle, as shown in Fig. 3.1. Even if the cycle is large and has many states, the sender only has to be able to cause one of the transitions in the cycle to occur—the sender just has to wait a sufficient length of time for the other transitions to occur and for the cycle to come around again. It is even better if the receiver can cause some of the other transitions in the cycle. Even in a large cycle, only two of the states of the cycle have to differ with respect to the receiver.

3.5.3 Encoding Schemes

There are several ways of encoding the data to be sent. One way is to keep one of the states stable for a fixed number of clock ticks and use the length of time between state changes as the signalling alphabet. Another way is to treat one complete cycle as a bit, similar to a Morse code pulse. It is possible to calculate the bandwidth of the covert channel in terms of the number of possible encodings of the data and the likelihood of each symbol over time [26, 27]. It is important to note that not all possible covert channels can be exploited at runtime, and further analysis may be needed to eliminate false positives.

3.5.4 Covert Storage Channel Detection

A straightforward static technique for analyzing the policy to detect possible covert storage channels involves first using a topological sort to determine if the graph has any cycles. Any core that can cause a transition in the cycle is a possible sender, and any core that can observe a difference between any two nodes is a possible receiver. This detector has proven itself on a wide range of policies, and the following pseudocode describes the detection algorithm:

```
Procedure DetectChannels (Graph G)
{
 Array of Lists Senders
 Array of Lists Receivers
 If (Topological_Sort(G) == False)
  Output ``Graph G Contains No Cycles.''
  Return
 C = Recursively_Trace_Graph_to_Find_Cycles(G)
 For (All Cycles C)
  For (All Edges E in C)
   M = Module that causes transition E
   Add M to Senders[C]
  For (All Vertices V in C)
    For (All Vertices V' in C) if V' != V then
     For (All Rows r)
      For (All Columns c)
       If (Matrix(V)[r][c] != Matrix(V')[r][c])
        Add c to Receivers[C]
 Output ``Possible Covert Channels:''
 For (All Cycles C Found)
  Output Cross_Product(Senders[C], Receivers[C])
 Return
}
```

Matrix(V) is the access matrix at node V. *Senders*$[C]$ is a list of modules that can cause transition E to occur within cycle C. *Receivers*$[C]$ is a list of modules that are able to observe a difference between the access matrices of any two nodes V and V' within cycle C.

3.5.5 Covert Channel Mitigation

Once a possible covert channel is identified at the policy level, the best option in terms of reducing risk is to revise the policy to eliminate the problematic cycle. If this is not feasible (for example, if critical services would have to be disabled), the system designer can limit the bandwidth of the channel by measuring it with a counter that keeps track of the number of times the cycle occurs within a sliding window of time. If the bandwidth exceeds a threshold during the measurement window, the system changes to a policy that does not have the problematic cycle. If needed, the system can revert to the old policy after a period of time. Another option is to add delays to transitions between states to throttle the bandwidth of the channel.

References

1. O. Aciíçmez, Yet another microarchitectural attack: exploiting I-cache, in *Proceedings of the First Computer Security Architecture Workshop (CSAW)*, Fairfax, VA, November 2007
2. O. Aciíçmez, S. Gueron, J.P. Seifert, New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures. IACR Cryptology ePrint Archive, Report 039, 2007
3. O. Aciíçmez, J.P. Seifert, Cheap hardware parallelism implies cheap security, in *Proceedings of the Fourth Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, Vienna, Austria, September 2007
4. O. Aciíçmez, J.P. Seifert, C.K. Koc, Micro-architectural cryptanalysis. IEEE Secur. Priv. **5**(4), 62–64 (2007)
5. S. Adee, The hunt for the kill switch. IEEE Spectrum **45**(5), 35–39 (2008)
6. R. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems* (Wiley, New York, 2001)
7. R. Anderson, M. Kuhn, Tamper resistance: a cautionary note, in *Proceedings of the Second USENIX Workshop on Electronic Commerce*, Oakland, CA, November 1996
8. Z. Chen, P. Schaumont, Slicing up a perfect hardware masking scheme, in *Proceedings of the 2008 IEEE International Workshop on Hardware-Oriented Security and Trust (HOST-2008)*, Anaheim, CA, June 2008
9. Defense Science Board, High performance microchip supply. *White Paper*, February 2005
10. K. Gandolfi, C. Mourtel, F. Olivier, Electromagnetic analysis: concrete results, in *Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, Paris, France, May 2001
11. I. Hadzic, S. Udani, J. Smith, FPGA viruses, in *Proceedings of the Ninth International Workshop on Field-Programmable Logic and Applications (FPL'99)*, Glasgow, UK, August 1999
12. W.M. Hu, Lattice scheduling and covert channels, in *Proceedings of the 1992 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1992
13. T. Huffmire, T. Sherwood, R. Kastner, T. Levin, Enforcing memory policy specifications in reconfigurable hardware. Comput. Secur. **27**(5–6), 197–215 (2008)
14. J. Kelsey, B. Schneier, C. Hall, D. Wagner, Side channel cryptanalysis of product ciphers. J. Comput. Secur. **8**(2–3), 141–158 (2000)
15. R.A. Kemmerer, Shared resource matrix methodology: an approach to identifying storage and timing channels, in *ACM Transactions on Computer Systems*, 1983
16. R.A. Kemmerer, A practical approach to identifying storage and timing channels: twenty years later, in *Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC)*, Las Vegas, Nevada, USA, December 2002
17. S.T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, Y. Zhou, Designing and implementing malicious hardware, in *Proceedings of the First Usenix Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, San Francisco, CA, April 2008
18. D. Kirovski, M. Drinic, M. Potkonjak, Enabling trusted software integrity, in *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, San Jose, CA, October 2002
19. P. Kocher, Timing attacks on implementations of Diffie-Hellman, RSA, DSS and other systems, in *Proceedings of the 16th Annual International Cryptology Conference (CRYPTO)*, Santa Barbara, CA, August 1996

20. P. Kocher, J. Jaffe, B. Jun, Differential power analysis, in *Proceedings of the 19th Annual International Cryptology Conference (CRYPTO)*, Santa Barbara, CA, August 1999
21. B.W. Lampson, A note on the confinement problem. Commun. ACM **16**(10), 613–615 (1973)
22. D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, M. Horowitz, Architectural support for copy and tamper resistant software, in *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, San Jose, CA, October 2000
23. J.I. Lieberman, National security aspects of the global migration of the US semiconductor industry. *White Paper*, June 2003
24. J. Lotspiech, S. Nusser, F. Pestoni, Broadcast encryption's bright future. IEEE Comput. **35**(8), 57–63 (2002)
25. J.P. McGregor, R.P. Lee, Protecting cryptographic keys and computations via virtual secure coprocessing, in *Workshop on Architectural Support for Security and Antivirus (WASSA) Held in Conjunction with the Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)*, Boston, MA, October 2004
26. J.K. Millen, Covert channel capacity, in *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, Oakland, CA, USA, April 1987
27. J.K. Millen, Finite-state noiseless covert channels, in *Proceedings of the Computer Security Foundations Workshop II*, Franconia, NH, USA, June 1989
28. D.A. Osvik, A. Shamir, E. Tromer, Cache attacks and countermeasures: the case of AES (extended version). Technical Report, Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot 76100, Israel, October 2005
29. D. Page, Theoretical use of cache memory as a cryptanalytic side-channel. Technical Report CSTR-02-003, Department of Computer Science, University of Bristol, June 2002
30. D. Page, Partitioned cache architecture as a side channel defense mechanism. *Cryptology ePrint Archive*, Report 2005/280, 2005
31. H.G. Rice, Classes of recursively enumerable sets and their decision problems. Trans. Am. Math. Soc. **74**, 358–366 (1953)
32. B. Sharkey, TRUST in integrated circuits program: briefing to industry, 26 March 2007. http://www.darpa.mil/MTO/solicitations/baa07-24/Industry_Day_Brief_Final.pdf
33. S.W. Smith, S.H. Weingart, Building a high-performance, programmable secure coprocessor. Comput. Netw. Int. J. Comput. Telecommun. Netw. (Spec. Issue Comput. Netw. Secur.) **31**(9), 831–860 (1999)
34. F. Standaert, L. Oldenzeel, D. Samyde, J. Quisquater, Power analysis of FPGAs: how practical is the attack? Field-Program. Log. Appl. **2778**(2003), 701–711 (2003)
35. N. Topham, A. Gonzalez, Randomized cache placement for eliminating conflicts. IEEETC: IEEE Trans. Comput. **48**, 185–192 (1999)
36. S. Trimberger, Trusted design in FPGAs, in *Proceedings of the 44th Design Automation Conference*, San Diego, CA, USA
37. Z. Wang, R. Lee, New cache designs for thwarting cache-based side channel attacks, in *Proceedings of the 34th International Symposium on Computer Architecture (ISCA)*, San Diego, CA, June 2007
38. X. Wang, M. Tehranipoor, J. Plusquellic, Detecting malicious inclusions in secure hardware: challenges and solutions, in *IEEE Workshop on Hardware Oriented Security and Trust (HOST)*, Anaheim, CA, June 2008
39. J. Yang, Y. Zhang, L. Gao, Fast secure processor for inhibiting software piracy and tampering, in *Proceedings of the Thirty-Sixth International Symposium on Microarchitecture (MICRO-36)*, San Diego, CA, December 2003
40. P. Yu, P. Schaumont, Secure FPGA circuits using controlled placement and routing, in *Proceedings of the 2007 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'07)*, Salzburg, Austria, October 2007

Chapter 4
FPGA Updates and Programmability

Abstract This chapter explains the security issues related to the programmability of FPGAs. FPGAs have the ability to change part or all of their configuration during runtime in the field. This chapter also explains how to prevent attackers from exploiting these features.

4.1 Introduction

Unlike an ASIC, an SRAM FPGA can change its logic configuration after it has been manufactured. The bitstream defining that logic is stored in non-volatile off-chip memory and is loaded onto the FPGA when the FPGA is powered on. This is useful because if a mistake is found in the logic design, the flawed bitstream can be replaced with a new bitstream at a small cost relative to fabricating an entirely new chip. In addition, it is not necessary to put *sensitive intellectual property* (referred to here simply as *IP*) at risk by sending it to a foundry that may not be entirely trusted. Instead, the bitstream may be loaded onto the FPGA in a secure facility after the FPGA has been manufactured.

4.2 Bitstream Encryption and Authentication

Storing a proprietary bitstream in non-volatile off-chip memory poses a security problem. The FPGA industry has invested considerable effort to develop bitstream encryption mechanisms to prevent the design from being extracted from the non-volatile memory. A symmetric cipher is used to encrypt the bitstream to be stored in the non-volatile memory. This prevents a board-level probing attack to read the bitstream as it travels over a bus from the non-volatile memory to the FPGA. The decryption process occurs on the FPGA. In theory, stealing the design would require an expensive, invasive *sand-and-scan* attack that involves physical methods to reverse engineer the chip [1].

T. Huffmire et al., *Handbook of FPGA Design Security*,
DOI 10.1007/978-90-481-9157-4_4, © Springer Science+Business Media B.V. 2010

First proposed in a patent by Austin [2], bitstream encryption relies on a cipher and a key to encrypt the bitstream and store it in non-volatile off-chip memory. Also, digital watermarking embeds a hidden signature that can be used to substantiate claims of design theft. One watermarking scheme for FPGAs works at the physical level by manipulating unused parts of the bitstream [8]. Another option for protecting the bitstream is to provide continuous power to a fielded SRAM FPGA after programming it in a secure facility. Removing the power will erase the secret keys used to encrypt the bitstream. Although it eliminates the need for bitstream encryption, continuous power is expensive, and reprogramming is not an option. Antifuse FPGAs, which are nonvolatile, do not require continuous power, although they cannot be reprogrammed. Flash-based FPGAs do not require continuous power, but they can be reprogrammed.

In addition to encryption, authentication is needed to prove the identity of the author of a bitstream or the identity of the person uploading a bitstream to a fielded device. Authentication can also be used to prevent IP from running on unauthorized FPGAs. Physical unclonable functions (PUFs) [13] are the basis of one scheme for the mutual authentication of IP and hardware [11]. The PUFs are used to generate a unique key for each chip by exploiting the slight manufacturing variations between chips. A third party uses the PUF and the identity of the author of the IP core to initiate a protocol for authenticating the hardware and software.

4.2.1 Key Management

In practice, however, key management is a significant concern. The key that is used to decrypt the bitstream needs to be stored somewhere. If it is hard-wired into the chip during manufacture then there are two possibilities: every chip that comes off the assembly line has the same key, or every device has a unique key. If they all have the same key, management of the keys introduces complexity, and it is possible to compromise every device by compromising one chip. If every device has a unique hard-wired key, then the database of keys for all of the devices will be a high priority target for attackers. The design choices include the following:

- Does each device have a unique key?
- Can the key be changed?
- What storage technology is used to store the key (EEPROM, battery-backed SRAM, etc.)?
- If a fixed key is compromised, is there a reserve key available, or does the device need to be discarded?
- Is they key symmetric or asymmetric? What cipher is used?
- What is the entropy of the key?
- Is there a key management infrastructure?
- How is the bitstream encryption mechanism implemented on the device? Where is it located?
- Is the decryption module susceptible to power analysis or timing analysis attacks?

Some devices have a unique key that can be set once after manufacture. Another option is to use Physical Unclonable Functions (PUFs). However, it is challenging to implement PUFs to reliably generate a sufficiently long key.

The Actel 60RS family of SRAM FPGAs employed an unfortunate scheme in which all devices shared the same key [9]. A fixed key was added to the FPGA during fabrication. While it protected against reverse-engineering of the design, it did not prevent *cloning*. Furthermore, discovery of the key affected *all* designs, and key information was embedded in the CAD tools. The Xilinx Virtex II family stored the key in continuously powered volatile memory, but each device had a unique key. Since the register containing the key was small, a battery could last for years [15]. The Virtex family disables readback, prevents reading and writing of the key, and checks the integrity of the bitstream.

Design Tip: *Key Management*. Just because a design uses crypto doesn't mean that it is secure. Keys must have sufficient entropy, and they must be managed properly. Your security analysis should take into account the key management infrastructure, whether it's a public key infrastructure (PKI) based on digital certificates or something more basic, such as a hard-wired symmetric key. Flaws in the crypto protocols or in the implementation of the PKI can be exploited by attackers. The analysis should take into account how the specific FPGA version being used implements bitstream encryption, since a flaw in this implementation can be exploited by adversaries. What cipher is used? Where is the decryption mechanism located? Does every device have the same key, do *classes* of devices share a key, or does every device have a unique key? Is the key hard-wired, is it stored in battery-powered volatile memory, is it stored in non-volatile memory, or is it based on a PUF? What is the entropy of the key? How difficult is it to perform a timing or power analysis attack on the decryption module? Is the decryption module hard-wired or implemented in reconfigurable logic?

4.2.2 Defeating Bitstream Encryption

Another concern is power analysis attacks on the bitstream decryption mechanism in the FPGA. This attack analyzes the power consumption of the crypto circuitry in order to systematically determine the key. This can be countered by adding a complementary circuit that *masks* the power consumption of the crypto circuit. Of course, another option for circumventing the bitstream encryption mechanisms is to use a social engineering attack to bribe an employee of one of the FPGA companies to divulge the keys or details of the security mechanisms. Protecting high-value data against determined adversaries requires consideration of the strength of the bit-stream encryption mechanisms, which may be strong enough for common commercial use but might not be sufficient for systems that handle high-value data. For these applications, anti-fuse FPGAs may be a better alternative than SRAM FPGAs, since bitstream encryption is not required because the logic never leaves the device.

4.3 Remote Updates

The ability to remotely apply patches and upgrades to the bitstreams of fielded FPGAs poses similar security problems to applying software patches to networked PCs. Many of the information security issues are very similar, such as replay attacks and man-in-the-middle attacks.

4.3.1 Authentication

In the software world, an attacker who gains control of an organization's software update mechanism can insert malicious software into a large number of systems. Denial-of-service (DoS) attacks can also occur if the attacker reprograms, turns off, or destroys an FPGA serving as a network router. Although an FPGA can be programmed locally, some FPGAs can be updated remotely through the network. FPGAs with remote update capability must have authentication mechanisms to ensure that only authorized administrators can change the configuration of the FPGA, and the updates must be delivered securely.

Authentication schemes include passwords, biometrics, and tokens (here, we echo the well-known notion of *something you know, something you are, or something you have*). Since each scheme has its advantages and limitations, selection of a scheme for the authentication of the administrator of a remotely-updated FPGA should consider which scheme best protects the system against unauthorized updates. For example, if a password-based scheme is used, attackers will exploit the fact that many users select passwords that are easy to guess. Furthermore, if an administrator uses the same username and password for multiple FPGAs, compromising one device compromises them all. Protecting the salted hash value of the administrator's password is essential to prevent a password cracking attack. Biometrics, on the other hand, require physical access to the system, add cost, and must strike a balance between false positives and false negatives, as well as addressing replay vulnerabilities.

> *Design Tip*: *Passwords*. Users, developers, and administrators must select passwords of sufficient entropy, and they should be required to change passwords periodically. User accounts must be managed properly, and unusual behavior, such as multiple failed login attempts, should result in an appropriate response.

Design Tip: *Authentication*. Authentication for remote updates requires careful consideration. Think about using more than one authentication scheme, and analyze their tradeoffs. Given their relative costs, what is the best allocation of security dollars for authentication? What are the design, implementation, testing, development, configuration, operation, maintenance, and audit requirements?

4.3.2 Trusted Recovery

For systems that protect high-value data, the benefits of remote updates must be weighed against the risks. If a system were compromised by attacking this update mechanism, it will be necessary to implement a trusted recovery mechanism to restore the system to a secure state. Accomplishing this in a timely and efficient manner will be necessary to ensure the availability of critical services.

4.4 Partial Reconfiguration

Some FPGAs have the ability to change part of their configuration during runtime. This *dynamic partial reconfiguration* (a.k.a. *partial reconfiguration*) ability allows for the logic of one core to be swapped out and replaced with the logic of an entirely different core. This space-saving feature is useful for designs in which area is a major constraint.

4.4.1 Applications of Partial Reconfiguration

Since swapping cores in and out increases the complexity and design cost, it is not widely used—Moore's Law doubles the area budget every two years anyway. However, there are a few exceptions, such as a reconfigurable crossbar switch [10]. Building a full crossbar on a single chip that connects N nodes to each other requires an amount of area proportional to N^2. However, if at any given time only a subset of the nodes need to be connected, then dynamic reconfiguration can be used to implement the equivalent of a full crossbar with less area.

Design Tip: *Partial Reconfiguration*. In general, partial reconfiguration increases design complexity, which makes security analysis more challenging.

4.4.2 Hot-Swappable vs. Stop-the-World

Systems that use partial reconfiguration can be divided into two categories: those that stop running when a core is swapped out for a different core and those that continue to operate during the swap. The second category of systems is referred to as *hot-swappable*. Hot-swappable systems are even more difficult to engineer because of the problem of mapping the old data and state to the new core. Typically, hot-swappable systems require two cores—one is the *active* core that runs while the *inactive* core is updating. Once the update is complete, the system immediately switches over, and the inactive core becomes the active core.

4.4.3 Internal Configuration Access Port

Partial reconfiguration uses the Internal Configuration Access Port (ICAP). One *frame* or *row* of the FPGA's configuration is read out from the ICAP. Part of this frame is then modified before being written back through the ICAP. This process can be driven by a processor core. Clearly, an attacker could obtain the entire bitstream just by reading it out of the ICAP. Therefore, most FPGAs disable bitstream decryption when partial reconfiguration is used so that an attacker cannot read the bitstream in the clear. Designers must consider the details of these mechanisms when designing systems that will process high-value data.

> *Design Tip*: *ICAP*. Although enabling partial reconfiguration disables bitstream decryption mechanisms to prevent the theft of an encrypted bitstream via the ICAP interface, a reconfigurable crypto core can be used together with the ICAP interface to protect the bitstream [6].

4.4.4 Dynamic Security and Complexity

Clearly, partial reconfiguration makes the engineering analysis more complex. In addition, the security analysis becomes more challenging as well, since partial reconfiguration increases the complexity of the system, especially when that can be used to change the security policy of the system. Building a system that enforces a single static policy is challenging enough, but a system that can change policies is a lot more complicated. The key design factors in dynamic security are:

- The number of policies from which the system can choose.
- Whether the policies are ad hoc (determined at runtime) or are predetermined and pre-validated.
- The frequency of policy changes.

- Whether it is possible to return to an earlier policy or to a *secure state*.
- Whether the system always transitions to a monotonically less restrictive policy.

Design Tip: *Hybrid Policies*. Policy changes require proper sanitization of cores in case there are lingering remnants of data or state to which a core should no longer have access. If your system requires dynamic security, make sure that only a trusted module can perform a policy change. Other key design factors include:

- The number of policies the system can choose from (limit this to minimize complexity). Significant analysis is required to ensure that the system maintains a secure state. As the number of possible policies increases, the amount of analysis increases.
- Whether the policies are predetermined (predetermined is preferable in order to simplify the security analysis).
- The frequency of policy changes (lower is better due to the risk of covert channels).
- Whether it is possible to return to an earlier policy (no is better in order to minimize the risk of covert channels).
- Whether the system always transitions to a monotonically less restrictive policy (yes is better due to the risk of data remanence).

4.4.5 Object Reuse

If a system changes to a more restrictive policy, it is necessary to *scrub* the memory and state of the system to remove any remnants of the data that has suddenly become illegal. *Secure object reuse* describes the secure reassignment of system resources to a processing element (e.g., a core) in a manner that prevents the new processing element from scavenging residual information inadvertently retained in the recycled resources.

In FPGAs that support partial reconfiguration, the designer doesn't want remnants of the previous core to linger because this could introduce unintended effects that could be exploited by an adversary. The use of a soft MicroBlaze processor core to read the configuration bitstream via the Xilinx ICAP interface one frame at a time is described in [7]. After reading in a frame, which spans the height of the device, part of the frame is erased, and the modified frame is written back to the device. This process can be done efficiently across a variety of devices. For an ICAP speed of 50 MHz, a single frame can be reconfigured on the order of ten microseconds, depending on the size of the device. Total reconfiguration time is proportional to the number of frames to be reconfigured.

Data Remanence occurs when attempts to achieve secure object reuse fail. Causes include the storage device's physical properties or removal operations that are incomplete. For example, DRAM cells retain their state for seconds to minutes after power is removed, even at room temperature. In a *cold boot* attack, compressed air is used to freeze the DRAM so that it can be removed from the motherboard and inserted into the attacker's computer, who then reads its contents (an example of a physical attack that calls for anti-tamper mechanisms like wire mesh) [5]. An example of data remanence involving incomplete removal operations is file deletion. Even when deleted, a file may remain on the hard drive until overwritten. Sensitive information may be compromised if a malicious party obtains the device. Techniques for dealing with data remanence on a disk include overwriting multiple times, degaussing, encryption, and physically destroying the device.

Data remanence in FPGAs is dependent on the design and the data since bits discharge at different rates depending on whether the value is one or zero [16]. Remanence also depends on whether the bits are related to logic or interconnect, since these two types have different supply voltages. Just like the cold boot example, FPGAs also retain charge longer at lower temperatures. Even when powered off, FPGAs can retain their bitstreams for up to two minutes. Analysis of the remanence of the bits makes it possible for attackers to obtain information about the plaintext bitstream. By grounding the power supply, remanence time can be reduced, and 20% information loss can occur within one millisecond [12].

4.4.6 Integrity Verification

Benjamin Glas et al. have developed a runtime method of checking the integrity of designs that use partial reconfiguration [4]. They use a special form of cryptographic hash functions called incremental hashing [3] to ensure that the FPGA configuration never enters a bad state. Incremental hashing solves the following problem: suppose Alice wants to compute the hash of a thousand page book. Alice's crypto hash function has to receive the entire book as input in order to compute the hash value. Now

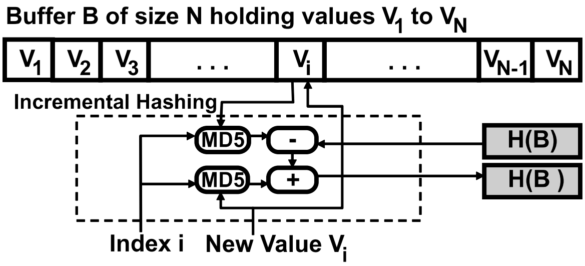


Fig. 4.1 Secure Incremental Hashing is a way to keep a cryptographically secure hash of an arbitrarily long buffer H(B), without requiring that the entire buffer be reexamined after a modification. Modifications are instead applied to the hash and the buffer at the same time. The hash of the original chunk is subtracted from H(B), and then the hash of the modified chunk is added to yield the new hash value H(B′)

suppose that Alice only changes one page of the book. Incremental hashing subtracts the hash of the old version of that page then adds in the hash of the modified version of that page. The resulting hash value is equal to the hash of the modified book, but Alice didn't have to read in the other 999 pages again. Figure 4.1 shows the incremental hashing process. Although not very useful for banking, there are a few examples of its use in computer architecture, including memory integrity verification in secure processors [14]. Glas et al. subtract the hash of the swapped out core and add in the hash of the swapped in core. They make sure that the hash value always belongs to the set of known good hash values. If the configuration enters a bad state, the system can mount a response.

Design Tip: *Dynamic Bitstream Integrity Verification*. Cryptographic hash functions can be used as the basis of schemes that verify the integrity of a bitstream, or parts of a bitstream, at runtime. One way to do this is to use a MicroBlaze soft processor core together with partial reconfiguration to drive the process. Bits of the configuration are read from the ICAP one frame of the time, and the hash value is computed on part or all of the bitstream. Since use of partial reconfiguration disables bitstream decryption, you will need to implement the bitstream decryption yourself [6]. Think carefully about whether this increase in design complexity is worth the cost. Does it expose the design to greater risk? How might an attacker defeat such an integrity verification scheme?

References

1. R. Anderson, M. Kuhn, Tamper resistance: a cautionary note, in *Proceedings of the Second USENIX Workshop on Electronic Commerce*, Oakland, CA, November 1996
2. K. Austin, Data security arrangements for semiconductor programmable devices. US Patent 5,388,157, February 1995
3. M. Bellare, D. Micciancio, A new paradigm for collision-free hashing: incrementality at reduced cost, in *Proceedings of Eurocrypt'97*, Konstanz, Germany, May 1997
4. B. Glas, A. Klimm, O. Sander, K. Müller-Glaser, J. Becker, A system architecture for reconfigurable trusted platforms, in *Proceedings of the 2008 Conference on Design Automation and Test in Europe (DATE'08)*, Munich, Germany, March 2008
5. J.A. Halderman, S.D. Schoen, N. Heninger, W. Clarkson, W. Paul, J.A. Calandrino, A.J. Feldman, J. Appelbaum, E.W. Felten, Lest we remember: cold boot attacks on encryption keys, in *Usenix Security Symposium*, San Jose, CA, July 2008
6. S. Harper, R. Fong, P. Athanas, A versatile framework for FPGA field updates: an application of partial self-reconfiguration, in *Proceedings of the 14th IEEE International Workshop on Rapid System Prototyping*, June 2003
7. T. Huffmire, B. Brotherton, G. Wang, T. Sherwood, R. Kastner, Moats drawbridges: an isolation primitive for reconfigurable hardware based systems, in *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2007
8. A.B. Kahng, J. Lach, W.H. Mangione-Smith, S. Mantik, I.L. Markov, M. Potkonjak, P. Tucker, H. Wang, G. Wolfe, Constraint-based watermarking techniques for design IP protection. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **20**(10), 1236–1252 (2001)

9. T. Kean, Secure configuration of field programmable gate arrays, in *Proceedings of the 11th International Conference on Field Programmable Logic and Applications (FPL'01)*, Belfast, UK, August 2001
10. P. Lysaght, D. Levi, Of gates and wires, in *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, Santa Fe, NM, April 2004
11. E. Simpson, P. Schaumont, Offline HW/SW authentication for reconfigurable platforms, in *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, Lausanne, Switzerland, September 2006
12. S. Skorobogatov, Low temperature data remanence in static RAM, Cambridge University Technical Report UCAM-CL-TR-536, ISSN 1476-2986, June 2002
13. G.E. Suh, S. Devadas, Physical unclonable functions for device authentication and secret key generation, in *Design Automation Conference (DAC)*, San Diego, CA, June 2007
14. G.E. Suh, B. Gassend, M. van Dijk, S. Devedas, Efficient memory integrity verification and encryption for secure processors, in *Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO-36)*, San Diego, CA, December 2003
15. S. Trimberger, Method and apparatus for protecting proprietary configuration data for programmable logic. US Patent 6,654,889, 2003
16. T. Tuan, T. Strader, S. Trimberger, Analysis of data remanence in a 90 nm FPGA, in *Proceedings of the IEEE Custom Integrated Circuits Conference (CICC)*, San Jose, CA, September 2007

Chapter 5
Memory Protection on FPGAs

Abstract This chapter describes a memory access policy language (Huffmire et al., Proceedings of the European Symposium on Research in Computer Security (ESORICS), Hamburg, Germany, September 2006), based on formal regular languages, and demonstrates how this language can express classical security policies, including isolation, controlled sharing, and Chinese wall. This chapter also describes a policy compiler (Huffmire et al., Proceedings of the European Symposium on Research in Computer Security (ESORICS), Hamburg, Germany, September 2006) that translates an access policy expressed in this language into a synthesizeable hardware module.

5.1 Overview

Managing external resources such as off-chip DRAM is essential to providing separation of the IP cores on an FPGA. Although general-purpose processor based systems often have virtual memory mechanisms with which to provide memory protection, FPGAs typically have a flat physical address space and a flat program structure that lacks control support (e.g., from an operating system). This condition makes it possible for a core to interfere with other cores by reading from or writing to their memory, whether from mistake or malice. To mitigate this problem, a *memory access policy* that all cores must obey is needed. The reconfigurability of FPGAs can be exploited to build a mechanism for enforcing these policies.

A *memory access policy* describes what memory accesses are legal or illegal (different policies take the negative vs. the positive approach), and a specialized language is used to formally describe the access policy. Being formally grounded facilitates the reasoning about policy soundness and the automatic refining of policies. A set of tools can automatically transform and synthesize the policy description to a circuit that functions as a monitor. This *reference monitor* bit-stream is loaded onto the FPGA along with the other cores, and it enforces the memory access policy by monitoring every memory access of every core and, based on its policy, allowing or disallowing each access.

T. Huffmire et al., *Handbook of FPGA Design Security*,

DOI 10.1007/978-90-481-9157-4_5, © Springer Science+Business Media B.V. 2010

The memory protection techniques presented in this chapter are steps towards a cohesive strategy for building trustworthy FPGA-based systems. For the embedded design community to accept this methodology, the design flow must generate high performance circuits that use the FPGA fabric efficiently. To pass muster with the security community, the resulting logic must enforce the policy faithfully and dependably. Finally, the techniques must be usable and understandable to both communities.

Design Tip: *Hardware Mechanisms for Policy Enforcement*. Not only must the enforcement mechanisms themselves be efficient in terms of area and timing, but they also must have minimal impact on overall system performance. Not only must the mechanisms faithfully and dependably enforce the policy locally, but the policy itself must also be correct. Usable and understandable tools for policy construction are helpful in ensuring policy correctness.

5.2 Memory Protection on FPGAs

A prototypical policy is that each core of an embedded system must be prevented from referencing or modifying memory belonging to another core without permission. There are several different kinds of memory, including on-chip block RAM, off-chip DRAM, and on- or off-chip backing-store such as Flash, all requiring efficient, flexible, and protected allocation and sharing. On general-purpose processor based systems, memory protection is generally based on page tables and *Translation Lookaside Buffers* (*TLBs*), and very large memory pages called *Superpages* have been proposed to give the TLB a lower miss rate [15]. However, providing per-process memory protection via global attributes is inefficient. In contrast, Segmented Memory [18] and the finer-grained Mondrian Memory Protection [22] associate each process with distinct permissions on the same memory region.

On modern FPGAs, which employ simple linear addressing, the memory is unprotected by hardware mechanisms. Even TLBs, a basic method of protection, are rarely found in embedded processors or reconfigurable devices. Although a TLB may speed up page table accesses, this comes at the cost of reduced system performance and more associative memory.

Memory protection is critical to preventing both errors and security attacks. Unfortunately, managing memory in software is not simple, and some subtle memory errors can be very difficult to detect. To provide memory protection for multiple concurrent *hardware* modules requires a different approach that borrows some key ideas from separation kernels. A separation kernel [17] uses virtualization to provide separation of software processes that is equivalent to physical separation. All resources are divided into partitions, which are isolated unless an explicit communication channel is created. In multilevel systems, each partition is assigned a level, and communication between partitions must obey the flow rules of the MLS-label lattice [7] projected onto the set of partitions.

This separation concept is applied to FPGAs by treating the cores and memory regions as partitions of a separation kernel. As Chap. 6 will discuss, the *moats* technique isolates the cores, and the *drawbridges* technique controls the interaction between cores securely [10]. To efficiently check every memory access, a reference monitor that recognizes a language of legal accesses is built in reconfigurable hardware, and all off-chip memory accesses are routed through it. Placing the enforcement module on the chip results in lower latency than placing it in a separate hardware module off-chip. In addition to protecting cores, the moats and drawbridges technique can provide isolation of the enforcement module and can also prevent it from being bypassed.

5.3 Policy Description and Synthesis

Although typical reconfigurable systems lack standard memory protection mechanisms, efficient memory protection mechanisms can be incorporated by exploiting the fine grain programmable properties of FPGAs. A compiler translates a memory access policy directly to a circuit that provides word-level stateful memory protection.

This section explains the memory access policy language and shows how to express a policy and compile it down to a synthesizeable description of a hardware module. In addition to describing the *formal top level specification* (*FTLS*), this section also explains the design flow that converts the policy into an enforcement module.

5.3.1 Memory Access Policy

A formal top level specification of a memory-access reference validation mechanism is based on system requirements and the organizational security policy [21]. The enforcement mechanisms described in this chapter are members of the Execution Monitoring (EM) class of enforcement mechanisms [19]. Members of this class monitor the execution of a *target*. On an FPGA, the target is the set of cores, as shown by the example in Fig. 5.1. Since the enforcement mechanism is a Reference Validation Mechanism (RVM), it must be protected, non-bypassable, and verifiable [3].

The formal top level specification precisely describes the set of legal memory access patterns to be recognized. This reference monitor must be able to track arbitrary size memory ranges, allowing only legal accesses and prohibiting all others. The language for the FTLS must be able to describe classical security policies, including isolation and controlled sharing, with a high level of clarity. This allows an engineer to express a policy in the language and to use a compiler to transform it to a

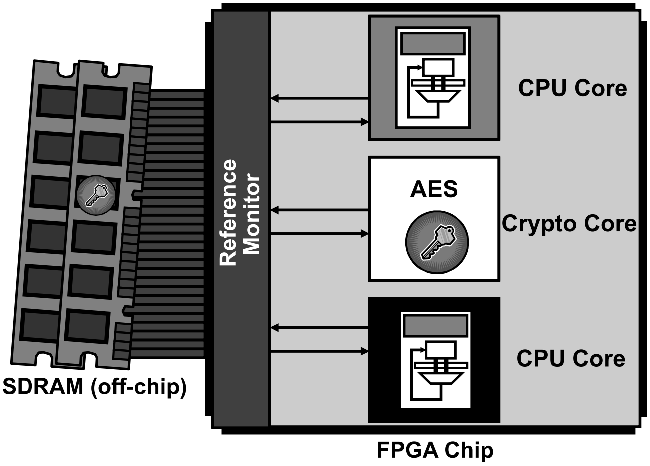


Fig. 5.1 An FPGA with two CPU cores and an AES crypto core, together with a reference validation mechanism (RVM), reside on an FPGA

regular expression. Applying regular languages provides a readable way to express policies. Established techniques can be applied for converting regular expressions to state machines and hardware [1]: thus automatically translating a high-level specification into an executable artifact.

We have developed two meta languages, a low-level meta language and a high-level meta-language. In the low-level meta language we recommend, the *Accessing Modules* (M) are the IDs of the cores on the chip. In security terminology, modules can also be referred to as subjects or principals. The *Access Methods* (A) describe *permissions*, which are the operations that a module can perform on an object, such as read, write, zero, etc. In the low-level language, the memory is partitioned into address *ranges*, which are the objects. The *Memory Range Specifier* (R) is the set of ranges that can be assigned a discrete permission value. Other ranges are reserved (e.g., for use by the RVM). The low-level language uses grammatical *productions* to describe an access policy, and each production specifies a relationship between modules (M), access rights (A), and ranges (R), where the element on the left-hand side of the arrow (→) is to be replaced by the expression on the right-hand side.

The low-level language uses *memory access descriptors* to specify a module's access right(s) to a range. These are the *terminals* of the productions of the low-level language, and they are tuples of the form (M, A, R). Although a memory access (M, A, k) involves a single address k, each memory access descriptor (M', A', R) is defined for a range of addresses. (M, A, k) is contained in (M', A', R) if and only if $M = M'$, $A = A'$, and $R_{low} \leq k \leq R_{high}$.

The space of all possible descriptor tuples is the cross product $\Sigma = M \times A \times R$. The formal definition of a memory access policy is a *regular* language, $L \subseteq \Sigma$, that defines a subset of possible descriptors and can be either finite or infinite. Enforcing the policy on an FPGA requires the ability to precisely define a policy notation for expressing L, an automatic way of generating a circuit that can recognize legal memory access sequences, and a way to prevent illegal accesses.

A simple example of an isolation policy helps to describe the recommended policy notation. Suppose that a system design has two modules, and each module may only access memory in its own specific range. For this scenario, the following productions illustrate the use of the low-level language:

$rw \rightarrow r|w;$
$Range_1 \rightarrow$ [0x8e7b008,0x8e7b00f];
$Range_2 \rightarrow$ [0x8e7b018,0x8e7b01b];
$Access_1 \rightarrow \{Module_1,\ rw,\ Range_1\};$
$Access_2 \rightarrow \{Module_2,\ rw,\ Range_2\};$
$Policy \rightarrow (Access_1|Access_2)^*;$

Policy is a non-terminal that defines the access policy (L) in terms of $Access_1$ and $Access_2$, which in turn are defined in the preceding productions. Although in this example $Access_1$ and $Access_2$ are very simple, using a formal grammar makes it possible to precisely compose more complex policies in a hierarchical fashion, with more complex sets of memory accesses.

The low-level language must be *regular* so that its sentences (those actions allowed by the policy) can be recognized by FSAs [14], which are efficient to implement in hardware. To make the job of constructing policies easier, it is not required that policies be expressed in right-linear or left-linear form. The compiler simply converts the policy from an extended regular grammar into a strictly regular grammar so that the notion of ranges can be expressed easily.

Address ranges can have arbitrary granularity, ranging from a unit of enforcement of a single word to the entire address space. This is very helpful for establishing shared *control words* that modules can use for secure coordination. Ranges are not allowed to overlap, and a static check verifies this. Despite the simplicity of the low-level language, it is versatile enough to express a wide variety of classical security policies, including isolation and Chinese wall. Now that the policy definition format has been described, the next section shows how a memory access policy can be automatically transformed into an efficient reconfigurable hardware module.

Design Tip: *Constructing a Policy*. The first step of policy construction is to specify the ranges. Next, specify the memory access descriptors. Each $Access_i$ can specify one or more access descriptors separated by the *OR* symbol (|). Finally, specify the policy in terms of the memory access descriptors. A *Policy* with only one state (i.e., a *stateless* policy) can simply be written as $(Access_1|Access_2|Access_3|\ldots|Access_n)^*$. A *stateful* policy with more than one state requires the specification of *Trigger* events (special terminals recognized by the language) that cause a transition from one state to another. Several examples of both stateful and stateless policies appear later in this chapter.

5.3.2 Hardware Synthesis

The policy compiler converts the grammar of an access policy into a circuit that is loaded onto the FPGA to enforce the policy at runtime. This circuit is divided into two parts: range detection hardware that identifies the range to which a given address belongs and the state machine that recognizes the language. The design flow consists of the following steps:

- The engineer writes the access policy, which is the input to the compiler.
- The compiler performs the following actions:
  - Constructs a syntax tree from the policy.
  - Converts the syntax tree to an expanded intermediate form.
  - Expands *Policy* to a regular expression.
  - Transforms the regular expression to a non-deterministic finite automaton (NFA).
  - Converts the NFA to a minimized deterministic finite automaton (DFA).
  - Converts each range into a covering set of aligned power of two ranges.
  - Outputs the range detection and state machine logic as Synthesizeable Verilog.
- Vendor synthesis tools (e.g., the Altera Quartus software) take the Verilog hardware description as input and synthesize, place, and route the circuit.
- The bit-stream loader loads the bit-stream onto the FPGA.
- During runtime, subject requests are allowed if their corresponding *descriptor* is accepted by the DFA.

5.3.2.1 Design Flow Details

Access Policy To illustrate how a simple policy is transformed to a circuit, recall that the isolation policy described earlier has two modules, and each module may only access its own range. The same policy can be expressed more concisely as follows:

$Access \rightarrow \{Module_1,\ rw,\ Range_1\}|\{Module_2,\ rw,\ Range_2\};$
$Policy \rightarrow (Access)^*;$

Parse Tree Construction and Transformation Next, the compiler uses Lex [13] and Yacc [12] to build a parse tree from the policy, shown in Fig. 5.2.

Next, the compiler substitutes the left hand side of each production with its right hand side, iteratively replacing all of the non-terminals. Figure 5.3 shows the transformed parse tree, from which the compiler can generate a regular expression.

Generating the Regular Expression Next, the compiler traverses the subtree corresponding to *Policy* to generate the regular expression. While in this simple example, it appears that the regular expression could easily be generated directly from the grammar, more complex policies require careful construction of the expanded parse tree:

$((\{Module_1,\ rw,\ Range_1\})|(\{Module_2,\ rw,\ Range_2\}))^*$

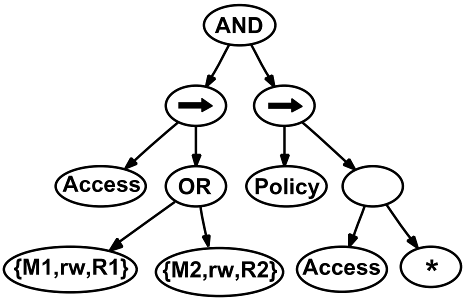


Fig. 5.2 Parse tree of the simple policy. *AND* is the root node, and each of its two children is a subtree corresponding to the two rules of the policy. *The right arrow symbol* is the root node of each of these subtrees

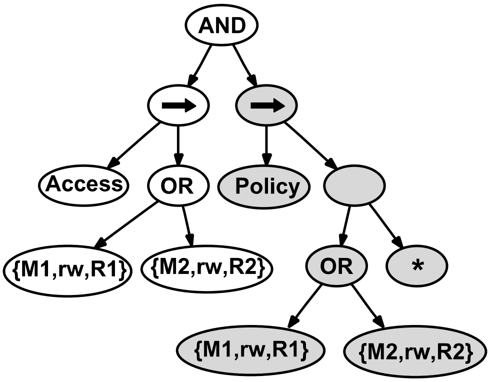


Fig. 5.3 Expanded parse tree. *The right subtree* corresponding to the second rule in the policy is shaded, and *Access* has been replaced with $\{Module_1,\ rw,\ Range_1\}$ $|\{Module_2,\ rw,\ Range_2\}$

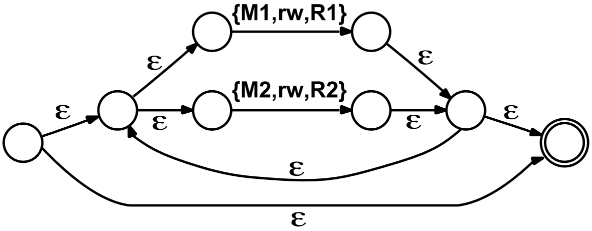


Fig. 5.4 NFA derived from the regular expression. Four epsilon transitions (top middle, closest to $\{M1,\ rw,\ R1\}$ and $\{M2,\ rw,\ R2\}$) are required for the *OR*. The other four epsilon transitions are for the Kleene star operation

NFA Construction Next, the compiler constructs an NFA from the regular expression using Thompson's Algorithm [1]. Figure 5.4 shows the NFA for the policy.

Converting the NFA to a DFA Next, the compiler uses subset construction to convert the NFA to a DFA [1]. Then, the compiler applies Hopcroft's Partitioning Algorithm [1] as implemented by Grail [16] to minimize the DFA. Figure 5.5 shows the minimized DFA for the policy.

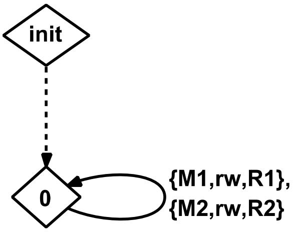


Fig. 5.5 NFA converted to a minimized DFA. Since this DFA only has one state, the policy that it enforces is referred to as a *stateless* policy

Processing the Ranges The compiler must process the ranges by converting them to a format that allows the circuit to efficiently compute the range containing a given address, searching the entire set of ranges in parallel. Using *don't care* bits, only the most significant bits need to be checked. For example, 10XX corresponds to 1000, 1001, 1010, and 1011, or [8, 11]. To check if 9 (1001) falls in this range, the hardware takes the bit-wise XOR of the first two bits of 1001 and 10XX. Since 10 XOR 10 is 00, 9 indeed falls in the interval [8, 11].

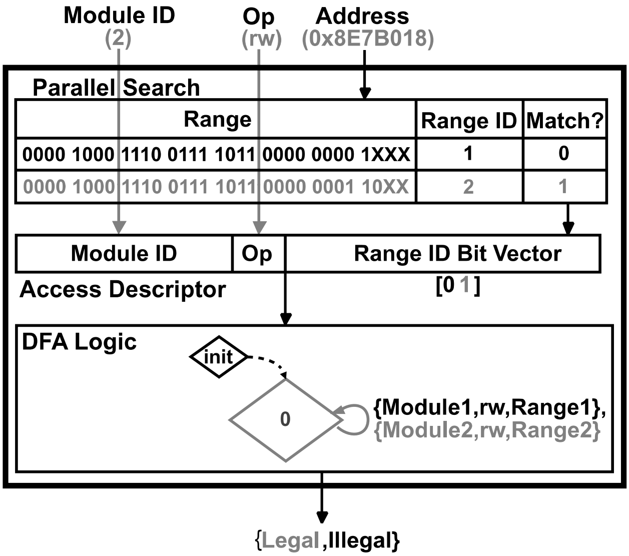
Any range whose size is a power of two can be described using *don't care* bits. The compiler must convert each range whose size is not a power of two into a covering set of $O(\log_2 |range|)$ ranges whose size is a power of two, where $|range|$ is the number of addresses in the range. For example, the range [7, 12] (0111, 1000, 1001, 1010, 1011, 1100) can be converted to the following set of ranges: {[7, 7], [8, 11], [12, 12]} (or {0111|10XX|1100}).

Conversion of the DFA to Verilog Because state machines are a very common hardware primitive, there are well-established methods of translating a DFA into a hardware description language such as Verilog. Figure 5.6 shows the architecture of the enforcement module. The inputs are the module ID, op, and address of the memory access, and the output is a single bit, 1 (grant) or 0 (deny). First, the hardware determines the range corresponding to the input address by checking all the ranges in parallel. Next, the DFA processes the request. If the DFA transitions to an accepting state, the output is 1, but if the DFA transitions to a rejecting state, the output is 0. If the DFA does not recognize the access request, the DFA transitions to the rejecting state. This is needed to prevent attackers from using undefined inputs as a way to cause the output to be 1.

State Machine Synthesis Finally, the Verilog code describing the enforcement module is converted to a bit-stream, which is loaded onto the FPGA along with the other cores. Vendor synthesis tools (e.g., the Altera Quartus design tools) synthesize, optimize, and perform place-and-route. Testing is required to verify the correct operation of the circuit.

5.4 A Higher-Level Specification Language

A policy enforcement mechanism such as a reference monitor is only as good as the policy it enforces. The meta language described in Sect. 5.3.1 is somewhat low-



Range	Range ID	Match?
0000 1000 1110 0111 1011 0000 0000 1XXX	1	0
0000 1000 1110 0111 1011 0000 0001 10XX	2	1

Fig. 5.6 The enforcement module takes three inputs: module ID, op, and address. A parallel search determines the range ID. The module ID, op, and range ID bit vector form an access descriptor, which the state machine logic uses to determine the output: 1 (grant) or 0 (deny)

level, requiring the embedded system designer to ponder complex regular expressions stated in terms of modules and ranges. A higher-level language will provide better usability by enabling policies to be expressed in terms of higher-level security concepts, reducing the risk of human error during the process of policy construction.

Research has shown that usability is critical to system security [23]. Security policies can be expressed at different layers of abstraction [21]. At the highest level is the organizational security policy [20], which is a document written in English that describes the security requirements of the organization. On the other hand, computer systems have specifications for, and mechanisms that enforce policies expressed at a much lower level of abstraction. Currently, completely ensuring the faithful translation from a high-level organizational security policy to lower-level implementations is an open problem (techniques for this are discussed in Sect. 2.6.7.3.

A higher-level language for expressing memory access policies increases the usability of the design flow. In this approach, an intermediate compiler translates a policy expressed in this higher-level language into the lower-level language described earlier in this chapter. Policies expressed in the higher-level language are significantly less complicated, especially for stateful policies such as Chinese wall, high water mark, and low water mark, which have exponential growth rates in the policy parameters (e.g., the number of conflict-of-interest classes, the size of the security label space, etc.) Expressing a high water mark or low water mark policy in the higher-level language only requires specifying the security labels of each module and range, and expressing a Chinese wall policy only requires specifying the elements of each conflict-of-interest class. Although the current implementation of our compiler has undergone some testing by the developer, further systematic testing will be required in order for the compiler to be production-grade. In other words, the implementation has a few bugs, and we inspect the output of each tool prior to using it in a design. Specifically, we simulate the circuit, applying a set of test inputs and validating the corresponding outputs before incorporating the circuit into a design. The higher-level language supports a variety of policies:

- Isolation
- Controlled sharing
- Access list
- Chinese wall
- Redaction
- Bell and LaPadula
- High water mark
- Biba
- Low water mark

The example isolation policy would be expressed as follows:

$Isolation$;
$Compartment_1 \rightarrow Module_1$;
$Compartment_1 \rightarrow Range_1$;
$Compartment_2 \rightarrow Module_2$;
$Compartment_2 \rightarrow Range_2$;

Notice that the first line of the file specifies the type of policy. In the higher-level language, the engineer specifies a set of compartments, and each compartment contains one or more modules and one or more ranges. A compartment with multiple modules is an equivalence class where the elements of the equivalence class are treated the same with respect to the policy.

5.5 Example Policies

Now that an isolation policy has been demonstrated, this section shows how to express other kinds of policies, including *stateful* policies involving revocation or conditional access. See [11] and [9] for more examples.

5.5.1 Controlled Sharing

Although the isolation policy prevents unintended flows of information between cores, sometimes cores need to be able to communicate with each other. The high- and low-level policy languages enable us to specify that one core can securely transfer data directly to another core without the need for intermediate communication buffers or multiple copies of the data. Instead, access to a specific range of data is transferred from one core to the next in a manner that prevents simultaneous access to or partial transfer of the data. For example, if *Module*$_1$ wants to securely transfer some data to *Module*$_2$, an access policy can synchronize the transition of permission to access a shared buffer. This policy is expressed in the lower-level language as follows:

$Access_1 \rightarrow \{Module_1,\ rw,\ Range_1|Range_3\}|\{Module_2,\ rw, Range_2\};$
$Access_2 \rightarrow \{Module_1,\ rw,\ Range_1\}|\{Module_2,\ rw, Range_2|Range_3\};$
$Trigger \rightarrow \{Module_1,\ rw, Range_4\};$
$Policy \rightarrow (Access_1)^*(\varepsilon|Trigger(Access_2)^*);$

> *Design Tip*: *Regular Expressions with Epsilon Term(s)*. Specifying the *Policy* expression is trickier for stateful policies. The ε term, which represents the empty string, is required because the policy might never make the transition (*Trigger*) to *Access*$_2$, instead happily staying in *Access*$_1$ forever.

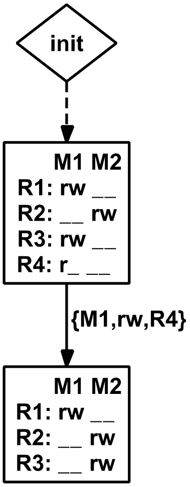
At first, *Module*$_1$ can access both *Range*$_1$ and *Range*$_3$ (the exchange buffer), but *Module*$_2$ can only access *Range*$_2$. *Module*$_1$ accesses *Range*$_4$ (the control word), indicating to the monitor that it is ready for the transition of permissions. This trigger event deactivates *Access*$_1$, revoking *Module*$_1$'s permissions to access *Range*$_3$. The trigger event also gives *Module*$_2$ exclusive access to *Range*$_3$.

In the higher-level language, the engineer specifies the *From* module, the *To* module, the exchange *Buffer*, and the *ControlWord*. Since controlled sharing has been implemented within the context of isolation, the engineer also specifies the compartments. The semantics of the higher-level language *CS* policy are that before the *from* module accesses the control word, only it can access the buffer, and after that, only the *to* module can access the buffer (in rw mode in both cases):

$CS;$
$From \rightarrow Module_1;$
$To \rightarrow Module_2;$
$Buffer \rightarrow Range_3;$
$ControlWord \rightarrow Range_4;$
$Compartment_1 \rightarrow Module_1;$
$Compartment_1 \rightarrow Range_1;$
$Compartment_2 \rightarrow Module_2;$
$Compartment_2 \rightarrow Range_2;$

Figure 5.7 shows the DFA that enforces this controlled sharing policy.

Fig. 5.7 The DFA corresponding to the controlled sharing policy



5.5.2 Access List

Sometimes a long list of subjects must have access to the same object. The access list policy is an isolation policy in which one or more modules belong to a list. The subjects in the policy are expressed in terms of lists rather than individual modules, much like a role or group. In the higher-level language, the designer first specifies the modules belonging to each list, and then expresses the policy in terms of these lists:

$AL;$
$List_1 \rightarrow Module_1;$
$List_1 \rightarrow Module_2;$
$List_1 \rightarrow Module_3;$
$List_1 \rightarrow Module_4;$
$List_2 \rightarrow Module_3;$
$List_2 \rightarrow Module_4;$
$Compartment_1 \rightarrow List_1;$
$Compartment_1 \rightarrow Range_1;$
$Compartment_2 \rightarrow List_2;$
$Compartment_2 \rightarrow Range_2;$

The higher-level specification above is translated to the following lower-level specification:

$$\begin{aligned} Access_1 \rightarrow\ & \{Module_1,\ rw,\ Range_1\}|\{Module_2,\ rw,\ Range_1\} \\ & |\{Module_3,\ rw,\ Range_1\}|\{Module_4,\ rw,\ Range_1\}; \\ Access_2 \rightarrow\ & \{Module_3,\ rw,\ Range_2\}|\{Module_4,\ rw,\ Range_2\}; \\ Policy \rightarrow\ & (Access_1|Access_2)^*; \end{aligned}$$

Figure 5.8 shows the DFA that enforces this access list policy.

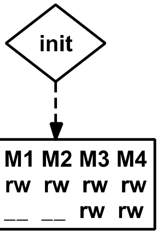


Fig. 5.8 The DFA corresponding to the access list policy

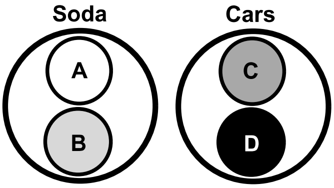


Fig. 5.9 This Venn Diagram shows two conflict-of-interest (COI) classes, one for soda companies and one for car companies. An attorney working on the Brand A Cola case cannot work on the Brand B cola case but may work on either the Brand C Auto case or the Brand D Auto case. To generalize this, e.g., in an embedded system, replace Brand A Cola, Brand B Cola, Brand C Auto, and Brand D Auto with $Range_1$, $Range_2$, $Range_3$, and $Range_4$

5.5.3 Chinese Wall

Another security scenario that can be efficiently expressed using the policy language is Chinese wall [6]. Suppose that $Company_1$ and $Company_2$ are competitors. They are said to belong to the same *conflict-of-interest* (*COI*) *class*. If a lawyer views the set of documents of $Company_1$, he or she should not view $Company_2$'s files. However, if $Company_3$ belongs to a different COI class than $Company_1$, the lawyer may view $Company_3$'s files. A Venn diagram illustrating this situation is shown in Fig. 5.9. In this example policy, $Module_1$ corresponds to the lawyer, and each range corresponds to a company:

$$Access_1 \rightarrow \{Module_1,\ rw,\ (Range_1|Range_3)\}^*;$$
$$Access_2 \rightarrow \{Module_1,\ rw,\ (Range_1|Range_4)\}^*;$$
$$Access_3 \rightarrow \{Module_1,\ rw,\ (Range_2|Range_3)\}^*;$$
$$Access_4 \rightarrow \{Module_1,\ rw,\ (Range_2|Range_4)\}^*;$$
$$Policy \rightarrow Access_1|Access_2|Access_3|Access_4;$$

This Chinese wall policy has two COI classes: one that contains $Range_1$ and $Range_2$ and another that contains $Range_3$ and $Range_4$. Figure 5.10 shows the DFA that enforces this policy.

A Chinese wall policy is expressed in the higher-level language by specifying the ranges that belong to each Conflict-of-Interest class ($Class_1$ or $Class_2$) as well as the module that is the subject in the policy:

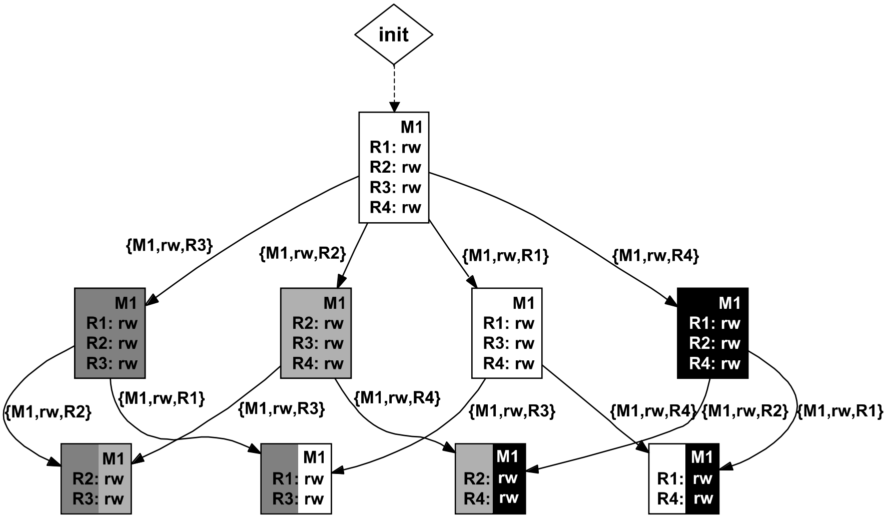


Fig. 5.10 This DFA enforces a Chinese wall policy. A core that accesses $Range_1$ (*white*) is subsequently prohibited from accessing $Range_2$ (*light gray*), but the core may access *either* $Range_3$ (*dark gray*) or $Range_4$ (*black*), which belong to a different conflict-of-interest class

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

5.5.4 Bell and LaPadula Confidentiality Model

The Bell and LaPadula (B&L) Model is a formal model of multilevel security in which a subject may not read an object with a higher security label (no read-up), and a subject may not write to an object with a lower security label (no write-down) [4]. This model is designed to protect the confidentiality of classified information. Assume that there are four labels in the label space: TOP SECRET (TS), SECRET (S), CLASSIFIED (C), and UNCLASSIFIED (U). The designer expresses a B&L policy in the higher-level language by specifying the security labels of each module and range:

B&L;
$Module_1 \rightarrow TS$;
$Module_2 \rightarrow U$;
$Range_1 \rightarrow U$;
$Range_2 \rightarrow U$;
$Range_3 \rightarrow TS$;

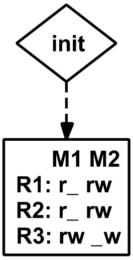


Fig. 5.11 The DFA corresponding to the B&L policy

The higher-level specification above is translated to the following lower-level specification:

$$\begin{aligned}Policy \rightarrow\ & (\{Module_1,\ r,\ Range_1\}|\{Module_1,\ r,\ Range_2\}\\ & |\{Module_1,\ rw,\ Range_3\}|\{Module_2,\ rw,\ Range_1\}\\ & |\{Module_2,\ rw,\ Range_2\}|\{Module_2,\ w,\ Range_3\})^*;\end{aligned}$$

Figure 5.11 shows the DFA that enforces this B&L policy.

5.5.5 High Water Mark

The high water mark model is an extension to B&L. High water mark is identical to B&L in that no read-up is permitted, but write-down is allowed in high water mark. Following a write-down, the security label of the object written to must change to the label of the subject that performed the write. Unlike B&L, high water mark policies are stateful. The designer expresses a high water mark policy in the higher-level language by specifying the security labels of each module and range:

$High$;
$Module_1 \rightarrow TS$;
$Module_2 \rightarrow U$;
$Range_1 \rightarrow U$;
$Range_2 \rightarrow U$;
$Range_3 \rightarrow TS$;

The higher-level specification above is translated to the following lower-level specification:

$$\begin{aligned}&Trigger_1 \rightarrow \{Module_1,\ w,\ Range_1\};\\ &Trigger_2 \rightarrow \{Module_1,\ w,\ Range_2\};\\ &Access_0 \rightarrow (\{Module_1,\ r,\ Range_1\}|\{Module_1,\ r,\ Range_2\}\\ &\qquad |\{Module_1,\ rw,\ Range_3\}|\{Module_2,\ rw,\ Range_1\}\\ &\qquad |\{Module_2,\ rw,\ Range_2\}|\{Module_2,\ w,\ Range_3\})^*;\end{aligned}$$

$$\begin{aligned}
&Access_1 \rightarrow (\{Module_1,\ rw,\ Range_1\}|\{Module_1,\ r,\ Range_2\}\\
&\qquad |\{Module_1,\ rw,\ Range_3\}|\{Module_2,\ w,\ Range_1\}\\
&\qquad |\{Module_2,\ rw,\ Range_2\}|\{Module_2,\ w,\ Range_3\})^*;\\
&Access_{12} \rightarrow (\{Module_1,\ rw,\ Range_1\}|\{Module_1,\ rw,\ Range_2\}\\
&\qquad |\{Module_1,\ rw,\ Range_3\}|\{Module_2,\ w,\ Range_1\}\\
&\qquad |\{Module_2,\ w,\ Range_2\}|\{Module_2,\ w,\ Range_3\})^*;\\
&Access_2 \rightarrow (\{Module_1,\ r,\ Range_1\}|\{Module_1,\ rw,\ Range_2\}\\
&\qquad |\{Module_1,\ rw,\ Range_3\}|\{Module_2,\ rw,\ Range_1\}\\
&\qquad |\{Module_2,\ w,\ Range_2\}|\{Module_2,\ w,\ Range_3\})^*;\\
&Access_{21} \rightarrow Access_{12};\\
&Path_1 \rightarrow (\varepsilon|Trigger_1 Access_1{}^*(\varepsilon|Trigger_2 Access_{12}{}^*));\\
&Path_2 \rightarrow (\varepsilon|Trigger_2 Access_2{}^*(\varepsilon|Trigger_1 Access_{21}{}^*));\\
&Policy \rightarrow Access_0{}^*(\varepsilon|Path_1|Path_2);
\end{aligned}$$

Design Tip: *Regular Expressions with Path Expression(s)*. This is a stateful policy with a *diamond* shape. There are two possible paths from the top of the diamond to the bottom: either through the right half of the diamond ($Path_1$) or through the left half of the diamond ($Path_2$).

Figure 5.12 shows the DFA that enforces this high water mark policy.

5.5.6 Biba Integrity Model

The Biba model is the dual of the Bell-LaPadula model [5]. Since Biba is designed to protect the integrity of data, both read-down and write-up are prohibited in Biba. One expresses a Biba policy in the higher-level language by specifying the integrity labels of each module and range:

$$\begin{aligned}
&Biba;\\
&Module_1 \rightarrow TS;\\
&Module_2 \rightarrow U;\\
&Range_1 \rightarrow U;\\
&Range_2 \rightarrow U;\\
&Range_3 \rightarrow TS;
\end{aligned}$$

Here, *TS* is high integrity, and *U* is low integrity. The higher-level specification above is translated to the following lower-level specification:

$$\begin{aligned}
&Policy \rightarrow (\{Module_1,\ w,\ Range_1\}|\{Module_1,\ w,\ Range_2\}\\
&\qquad |\{Module_1,\ rw,\ Range_3\}|\{Module_2,\ rw,\ Range_1\}\\
&\qquad |\{Module_2,\ rw,\ Range_2\}|\{Module_2,\ r,\ Range_3\})^*;
\end{aligned}$$

Figure 5.13 shows the DFA that enforces this Biba policy.

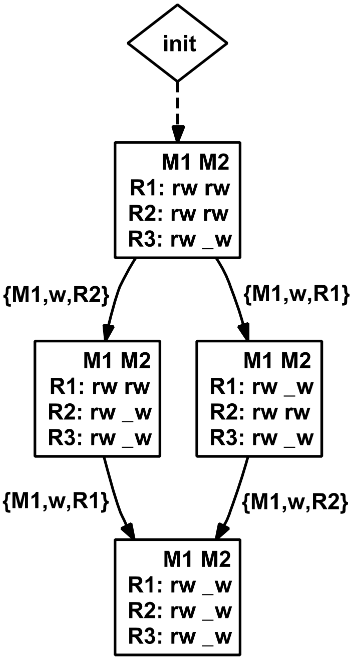


Fig. 5.12 The DFA corresponding to the high water mark policy

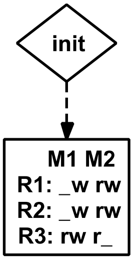


Fig. 5.13 The DFA corresponding to the Biba policy

5.5.7 Redaction

Redaction is the process of removing sensitive portions of a document so that someone with a lower clearance can read it. Consider the example of an FPGA system with three cores shown in Fig. 5.14 In this scenario, a multilevel database contains both classified and unclassified information. One core ($Module_1$) can read and write classified information and can read unclassified information. Another core ($Module_2$) is only cleared to read and write unclassified information, and a third core ($Module_3$) acts as a trusted server, retrieving information from the database in response to queries, performing redaction if necessary, and writing the data to a specific range of memory ($Range_3$). $Module_1$ and $Module_2$ perform database queries by writing to a control word ($Module_4$). If $Module_1$ performs a database query, $Module_2$ must be temporarily blocked from accessing $Range_3$ because classified in-

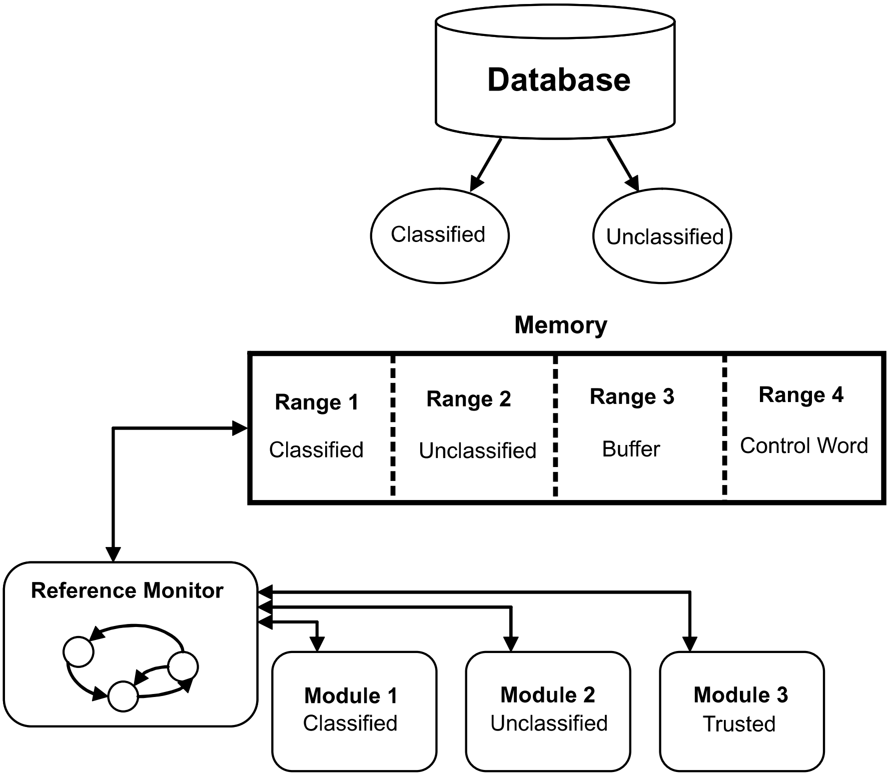
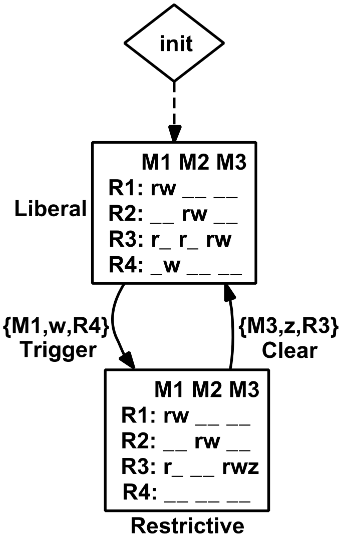


Fig. 5.14 Architecture of the redaction scenario

formation can be written there in response to the query. $Module_2$ must wait until the trusted server ($Module_3$) zeroes out $Range_3$. This policy consists of two states: a liberal state, in which $Module_2$ can access $Range_3$, and a restrictive state, in which $Module_2$ cannot access $Range_3$. When $Module_1$ performs a database query by writing to $Range_4$, this trigger event causes a change from the liberal to the restrictive state. When $Module_3$ zeroes out $Range_3$, this trigger event causes a change from the restrictive state back to the liberal state. In this example, redaction has been implemented in the context of isolation, so that only $Module_1$ may read from or write to $Range_1$, and only $Module_2$ may read from or write to $Range_2$. In the higher-level language, one specifies the liberal and restrictive policies, the trigger event, and the clear event:

$$
\begin{aligned}
&\mathit{Redaction};\\
&\mathit{Restrictive} \rightarrow \{\mathit{Module}_1,\ rw,\ \mathit{Range}_1\}|\{\mathit{Module}_1,\ r,\ \mathit{Range}_3\}\\
&\qquad\qquad |\{\mathit{Module}_2,\ rw,\ \mathit{Range}_2\}|\{\mathit{Module}_2,\ w,\ \mathit{Range}_4\}\\
&\qquad\qquad |\{\mathit{Module}_3,\ rw,\ \mathit{Range}_3\};
\end{aligned}
$$

Fig. 5.15 The DFA corresponding to the redaction policy



$Liberal \rightarrow Restrictive|\{Module_2,\ r,\ Range_3\};$
$Trigger \rightarrow \{Module_1,\ w,\ Range_4\};$
$Clear \rightarrow \{Module_3,\ z,\ Range_3\};$

The higher-level specification above is translated to the following lower-level specification:

$$\begin{aligned}
&Access_1 \rightarrow \{Module_1,\ rw,\ Range_1\}|\{Module_1,\ r,\ Range_3\}\\
&\qquad\quad |\{Module_2,\ rw,\ Range_2\}|\{Module_2,\ w,\ Range_4\}\\
&\qquad\quad |\{Module_3,\ rw,\ Range_3\};\\
&Access_2 \rightarrow Access_1|\{Module_2,\ r,\ Range_3\};\\
&Trigger \rightarrow \{Module_1,\ w,\ Range_4\};\\
&Clear \rightarrow\ \{Module_3,\ z,\ Range_3\};\\
&SteadyState \rightarrow\ (Access_1|Clear\ Access_2{}^*\ Trigger)^*;\\
&Policy \rightarrow \varepsilon|Access_2{}^*|Access_2{}^*\ Trigger\ SteadyState\\
&\qquad\quad |Access_2{}^*Trigger\ SteadyState\ Clear\ Access_2{}^*;
\end{aligned}$$

> *Design Tip*: *Simplifying Regular Expressions*. Since this DFA has cycles, a complex regular expression is required. To simplify the regular expression, common subexpressions can be substituted with nonterminals, such as *SteadyState* in this example.

Figure 5.15 shows the DFA that enforces this redaction policy.

5.6 System Architecture

The reference monitor must be placed strategically in the architecture in order to help provide the properties of nonbypassability and self-protection, as well as to minimize its impact on memory system performance. Systems vary widely in the number of cores, the manner in which the system elements communicate (direct connection, bus, or network), and the kinds of resources to be protected (on-chip BRAM and SRAM, off-chip DRAM, etc.) In a dual-core system in which modules access off-chip DRAM via a bus, a naive designer might put the enforcement mechanism between the bus and the memory, requiring every memory access to wait for the enforcement mechanism's approval before going to memory. The resulting delay is the sum of the memory latency and the time to approve the access. A smarter approach would be to have the enforcement mechanism snoop on the bus so that the memory access can occur in parallel with the task of approval. A buffer holds the data retrieved until approval is granted. For writes, the buffer stores the data to be written until approval is granted, although this does not alleviate the latency and approval time, unless a roll-back scheme is used. While both strategies provide the necessary isolation, they offer a tradeoff between performance and complexity.

An arbitration mechanism is needed to prevent modules from accessing the bus simultaneously. A simple approach would be to place an arbiter between each module and the bus, and the arbiters would restrict each core's use of the bus to its assigned time slice. Chapter 7 describes a multi-core embedded system that integrates both the arbiter and reference monitor into the bus. This design example consists of two MicroBlaze processors whose sharing of an AES encryption core is governed by a stateful policy.

Design Tip: *Placement of Enforcement Mechanisms*. The decision of where to place policy enforcement mechanisms not only affects security but also impacts performance, especially if the entire system has to wait for the reference monitor's decision before proceeding. Techniques for alleviating the delays, such as concurrency and pipelining, also introduce additional complexity.

5.7 Evaluation

Research [9] and [11] shows that the FPGA reference monitor is efficient in terms of area and performance. Circuit complexity depends on a combination of the number of memory ranges and the number of DFA states and transitions. To study the impact of the range detection task on system performance, an experiment involved varying the number of ranges in the isolation policy and synthesizing the resulting enforcement modules using Altera's Quartus synthesis tool [2]. The results of the experiment demonstrated a linear relationship between the size of the circuit and the number of ranges. The *setup time* to perform the range detection task also grows nearly linearly with the number of ranges, but this can be reduced with pipelining. On the Altera Stratix target device used in the experiment, the setup time varied from just over one cycle for a policy with a handful of ranges to six cycles for a policy with several hundred ranges. The *cycle time* for one DFA transition is nearly constant with the number of ranges. The average cycle time was approximately 6 ns, which is very close to the maximum speed of the target device.

FPGAs do not operate at a high clock rate, and 200 MHz is a common frequency that might not sound very impressive. Instead, they exploit computational parallelism to achieve high performance. Applications such as DSPs, face recognition, computer vision, and intrusion detection can exploit parallelism well, and they are not very sensitive to latency because they are throughput-driven. Since a 200 MHz FPGA has a cycle time of 5 ns, the reference monitor only adds a delay that is less than two cycles.

Design Tip: *Area, Setup Time, Cycle Time, and Throughput*. Setup time is the amount of time required to perform range detection, and cycle time is the amount of time for a single DFA transition. There is a linear relationship between the complexity of the policy and both the area and setup time of the reference monitor. Policy complexity does not affect cycle time, which is always around one cycle. Throughput, not latency, is the key performance metric of FPGA systems.

5.8 Using the Policy Compiler

Applying the design flow to a simple isolation policy illustrates its use. The first step is to create a file *toy.policy* with the following contents:

```
Isolation;
Compartment1->Module1;
Compartment1->Range1;
Compartment2->Module2;
Compartment2->Range2;
```

Note that the isolation policy has been expressed in the higher-level language.

Specifying the ranges: Ranges are specified in the file *ranges* as follows:

```
0000000 000000f
0000010 000001f
...
```

The first line of the file specifies $Range_1$, the second line of the file specifies $Range_2$, and so on. Each line of the file has the starting and ending address of the range separated by a space. Each range must be an aligned power of two. The next step is to use the compiler on this policy:

```
% ./run.sh toy
not found
0 is a start state
There are 1 unique states
This graph does NOT contain a cycle.
```

The script *run.sh* performs the following steps:

- *run.sh* processes the ranges.
- *run.sh* creates a file *toy.p* (the lower-level policy specification) from *toy.policy* (the higher-level policy specification).
- *run.sh* runs *toy.p* through the parser.
- The resulting regular expression is fed as input to RegEx, which creates a file *grail_machine*, which is a DFA expressed in Grail format. RegEx is an implementation of Thompson's Algorithm and subset construction by Gerzic [8], modified to output state machines in a format that is compatible with Grail.
- *run.sh* runs *grail_machine* through Grail to produce a file *gm_toy*, which is the minimized DFA. A slight modification to Grail allows it to handle *unsigned long integers*.
- *run.sh* creates from *gm_toy* a file *toy.v*, which is a Verilog HDL description of the reference monitor that enforces the policy.
- *run.sh* checks to see whether the DFA has any possible covert channels.
- *run.sh* creates a file *toy.dot*, which expresses the DFA in Graphviz format.
- *run.sh* runs *toy.dot* through Graphviz, resulting in a file *toy.ps*, which is a PostScript version of the DFA that can be printed or displayed on the screen. Figure 5.16 shows this graph.

The file *toy.p* is the lower-level policy specification generated from *toy.policy*, the higher-level policy specification:

```
Access0->{Module1,rw,Range1};
Access1->{Module2,rw,Range2};
Policy->(Access0|Access1)*;
```

The file *toy.v* is the Verilog HDL description of the reference monitor that enforces *toy.policy*:

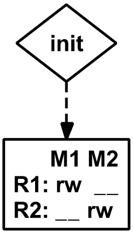


Fig. 5.16 The DFA corresponding to a *toy* isolation policy for demonstrating the design flow

```
module State_Machine(clock,
                     reset,
                     module_id,
                     op,
                     address,
                     is_legal);
   input clock, reset;
   input [4:0] module_id;
   input [1:0] op;
   input [31:0] address;
   output is_legal;
   reg is_legal;
   reg[0:0] state;
   parameter s0 = ’d0;
   parameter s1 = ’d1;
   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)
   begin
      case (state)
         s0:
            is_legal=1’b1;
         s1:
            is_legal = 1’b0;
         default:
            is_legal = 1’b0;
      endcase
   end
   always @(posedge clock or posedge reset)
   if (reset) state = s0;
   else
      case (state)
         s0:
            case({module_id,op,r0,r1})
               9’b000101101: //2 3 1
                  state = s0;
               9’b000011110: //1 3 0
                  state = s0;
               default:
                   state = s1;
            endcase
         s1:
            state = s1;
         default:
            state = s1;
      endcase
endmodule
```

The reference monitor expressed in the above Verilog code has three inputs (*module_id*, *op*, and *address*) and one output (*is_legal*). It uses exactly one cycle to make a decision as to the legality of the requested memory access according to the policy. *op* has four possible values: 00 for neither read nor write, 01 for read only, 10 for write only, and 11 for both read and write. The *assign* statements check the ranges in parallel to determine the range of *address*. The expression {*module_id*, *op*, $r0$, $r1$} concatenates *module_id*, *op*, $r0$, and $r1$ to form a single transition symbol. The first *case* statement determines the value of *is_legal* according to whether the state is accepting or rejecting. The second *case* statement determines the next state according to the current state and the transition character.

5.9 Constructing Mathematically Precise Policies

In order for a policy to be precise, it must accept all legal behavior and reject all illegal behavior. An automatic method is needed to check that the policy reflects the intent of the designer. Although this is a hard problem, the higher-level specification language helps to mitigate this problem, and this section describes how to check for conflicts in the policy between legal and illegal behavior.

5.9.1 Cross Product Method

One requirement of a correct policy is that the same behavior cannot be both legal and illegal. The result of taking the intersection between the language of legal accesses and the language of illegal accesses should be the empty set. Otherwise (if there is any overlap), it is necessary to notify the designer so that he or she can make corrections. Computing the intersection of two policies requires simply taking the cross product of their state machines using Grail (specifically the *fmcross* command). The cost of this computation is quadratic in the size of the state machines. The Venn diagram in Fig. 5.17 shows the basic idea.

Figure 5.18 is a Venn diagram showing an incremental approach of constructing policies. A *rough draft* of a specification of legal accesses is checked for any overlap between specific instances of known illegal behavior. This automated process can test a large space of known illegal behavior and notify the designer when there is any overlap.

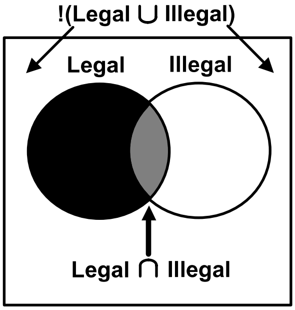


Fig. 5.17 A Venn diagram showing an incorrect policy where legal and illegal behavior incorrectly intersect

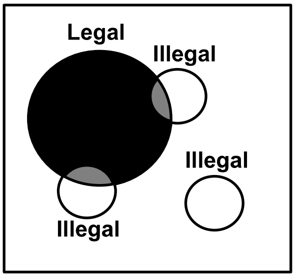


Fig. 5.18 A Venn diagram showing an automated approach to the incremental construction of policies. Several examples of known illegal behavior can be automatically checked against a *rough draft* specification of legal accesses to determine whether there is any intersection

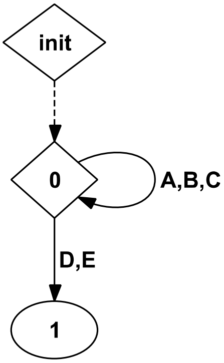


Fig. 5.19 DFA that recognizes the language $(A|B|C)^*$. An input of either D or E causes this DFA to transition to the rejecting state (State 1)

5.9.2 Examples

Consider the simple example of a language of legal behavior $L_{Legal} = (A|B|C)^*$ over the alphabet A, B, C, D, E. Figure 5.19 shows the DFA that accepts L_{Legal}. Suppose also that there is a language of illegal behavior $L_{Illegal} = (C|D|E)^*$. Figure 5.20 shows that DFA that accepts $L_{Illegal}$. Figure 5.21 shows the DFA that accepts $L_{Legal} \times L_{Illegal}$, which is C^*.

The same method can be used to compute the intersection of the B&L and Biba policies. Figure 5.22 shows the DFA for the B&L policy, except now the two extra transitions to the rejecting state (State 1) are explicitly shown. Figure 5.23 shows

Fig. 5.20 DFA that recognizes the language $(C|D|E)^*$. An input of either A or B causes this DFA to transition to the rejecting state (State 1)

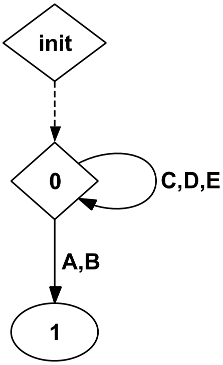


Fig. 5.21 DFA that recognizes the language C^*

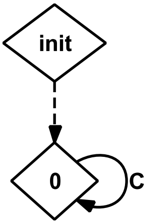
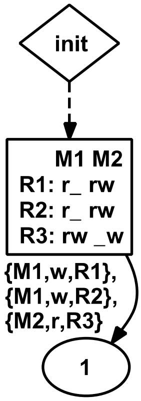


Fig. 5.22 This is the DFA for the B&L policy from Sect. 5.5.4. An input of $\{Module_1,\ w,\ Range_1\}$ or $\{Module_1,\ w,\ Range_2\}$ causes this DFA to transition to the rejecting state (State 1)



the DFA for the Biba policy, except now the two extra transitions to the rejecting state (State 1) are explicitly shown. Figure 5.24 shows the DFA that recognizes the intersection of the B&L and Biba policies, and it is computed by taking the cross product of their respective DFAs.

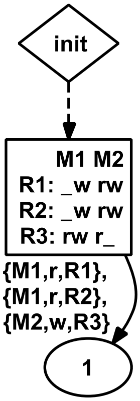


Fig. 5.23 This is the DFA for the Biba policy from Sect. 5.5.6. An input of $\{Module_1,\ r,\ Range_2\}$ or $\{Module_2,\ r,\ Range_2\}$ causes this DFA to transition to the rejecting state (State 1)

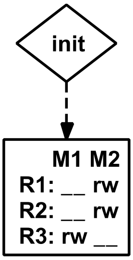
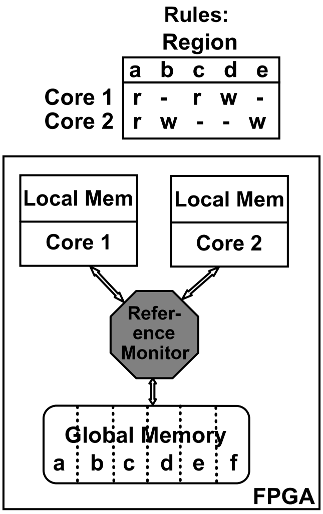


Fig. 5.24 This DFA recognizes the intersection B&L and Biba policies, such that they are both enforced, and it is computing by taking the cross product of their respective DFAs

5.9.3 Monotonic Policy Changes

The ability to determine the intersection of two policies is also useful for dynamic policies. In a system with the ability to switch policies dynamically, the cross product method can ensure that all policy changes are monotonic. For example, if the system changes from a less restrictive policy to a more restrictive policy, a core could retain sensitive information in its local memory after the new policy becomes effective. Although access to this data was legal under the old policy, the new policy prohibits it. A naive solution would be to sanitize all of the cores in the system following a policy change, but this is costly and could disrupt critical services. Another solution is to always change to a less restrictive policy. In a system that only allows changes to monotonically less restrictive policies, each policy is a superset of the previous policy. In other words, the intersection of $Policy_i$ and $Policy_{i+1}$ is identical to $Policy_i$. Suppose that a set of policies $\{Policy_1,\ Policy_2,\ Policy_3, \ldots,\ Policy_N\}$ is available in a dynamic policy system. To determine which policy changes are monotonic, the designer takes the intersection of every $(Policy_i,\ Policy_j)$ pair and checks if the result is identical to $Policy_i$. If so, a change from $Policy_i$ to $Policy_j$ is monotonically less restrictive.



Rules:
Region

	a	b	c	d	e
Core 1	r	-	r	w	-
Core 2	r	w	-	-	w

Fig. 5.25 FPGA Memory Control. An FPGA system consists of several processing cores (two in this example), each with local memory. The system also has global memory and interconnections between processing and memory elements. A reference monitor controls which global memory regions (a, b, c, d, e, f) that each core may access, as well as the mode of access: read or write

5.9.4 Formal Aspects of Hybrid Policies

As shown in Fig. 5.25, an FPGA system consists of several processing cores, each with local memory. The system also has on-chip global memory, off-chip global memory, and interconnections between processing and memory elements. A reference monitor controls which global memory regions that each core may access, as well as the mode of access: read and write. The governing security policy can be viewed as a table of rules.

The policy rules may change in response to a system event. Additionally, some organizational security policies include state changes triggered by actions of users, which result in the change of one or more rules. One problem with rule changes is that a core may have access to information in local memory that is acceptable to the current policy, but this access will be prohibited by the next policy. Since the reference monitor does not control access to local memory, a means of mitigating this kind of prohibited access is needed. One approach is to zero out the local memory of a problematic core as part of a policy change; another approach is to prohibit the type of policy changes that would cause the problem. Both of these require a precise understanding of what the problem is. The problem is stated in terms of global properties, which may be violated by a policy change.

The policy table P consists of a set of rules, $R = \{s,\ o,\ a\}$, indicating the allowed accesses, where s is an element from the set of subjects (i.e., cores), o is from the set of memory regions, and a is from the enumeration of access modes: null, read, write, read and write. Various operations are allowed on global and local memory regions. The primary operation on global memory will be abstractly represented as $g\_mem\_acc(s,\ o,\ a)$. For example, $g\_mem\_acc(s,\ o,\ r)$ transfers a global memory region into the local state (e.g., a register) of the subjects(s) (note that memory can easily be characterized at a more granular level, such as bytes, rather than regions).

Local memory is represented as $local(s)$. The read operation is represented this way, where the new value of a variable is indicated with the prime ($'$) symbol, and individual elements are indicated with subscripts.

$$g\_mem\_acc(s_1, o_1, r) \quad \Rightarrow \quad local'(s_1) = local(s_1) \cup o_1 \tag{5.1}$$

We assume that a read results in the data becoming part of local memory, because the subject may modify local memory based on the value of the data, even if it does not write the data directly to its local memory (alternatively, local memory could be modeled to include local registers).

The global security property we are interested in is:

$$\forall s : subject, o : object, a : (access(g\_mem\_acc(s, o, a)) \Rightarrow [s,\ o,\ a] \in P) \tag{5.2}$$

I.e., the subject can perform only the allowed accesses on global memory. A policy change is represented as:

$$P\_change(new : table) \quad \Rightarrow \quad P' = new \tag{5.3}$$

We need to ensure that whenever $P' \neq P$, the property (5.2) still holds. The following relationships will suffice (corresponding to the two approaches described above):

$$\begin{aligned} P' \neq P \quad \Rightarrow \quad & [\forall s : subject, o : object, \\ & a : access([s,\ o,\ a] \in P \rightarrow [s,\ o,\ a] \in P' | local'(s) = \emptyset)] \end{aligned} \tag{5.4}$$

I.e., if the policy changes then each access in the old policy is in the new policy, or the local memory of all subjects is cleared.

5.10 Summary

To prevent improper memory sharing and to contain memory bugs, this chapter has described a method and language for specifying access policies that can be automatically synthesized to a reconfigurable hardware enforcement module. The policy language facilitates the expression of policies of arbitrary granularity. To evaluate the efficiency of these techniques, experiments involved generating a variety of policies and using the design flow to synthesize hardware modules from the policies. These methods are efficient and scalable in the number of ranges that must be recognized. The next chapter shows how the architecture ensures that the enforcement module is both protected and invoked for every memory access.

References

1. A. Aho, R. Sethi, J. Ullman, *Compilers: Principles, Techniques, and Tools* (Addison Wesley, Reading, 1988)
2. Altera Inc, Quartus II Manual, 2004

3. J.P. Anderson, Computer security technology planning study. Technical Report ESD-TR-73-51, ESD/AFSC, Hanscorn AFB, Bedford, MA, 1972
4. D.E. Bell, L.J. LaPadula, Secure computer systems: mathematical foundations and model. The MITRE Corporation, Bedford, MA, USA, May 1973
5. K.J. Biba, Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, 1977
6. D.F.C. Brewer, M.J. Nash, The Chinese wall security policy, in *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, 1989
7. D.E. Denning, A lattice model of secure information flow. Commun. ACM **19**(5), 236–243 (1976)
8. A. Gerzic, CodeGuru: write your own regular expression parser, November 2003, http://www.codeguru.com/
9. T. Huffmire, S. Prasad, T. Sherwood, R. Kastner, Policy-driven memory protection for reconfigurable hardware, in *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, Hamburg, Germany, September 2006
10. T. Huffmire, B. Brotherton, G. Wang, T. Sherwood, R. Kastner, Moats and drawbridges: an isolation primitive for reconfigurable hardware based systems, in *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2007
11. T. Huffmire, T. Sherwood, R. Kastner, T. Levin, Enforcing memory policy specifications in reconfigurable hardware. Comput. Secur. **27**(5–6), 197–215 (2008)
12. S. Johnson, Yacc: yet another compiler-compiler. Technical Report CSTR-32, Bell Laboratories, Murray Hill, NJ, 1975
13. M. Lesk, E. Schmidt, Lex: a lexical analyzer generator. Technical Report 39, Bell Laboratories, Murray Hill, NJ, October 1975
14. P. Linz, *An Introduction to Formal Languages and Automata* (Jones and Bartlett, Sudbury, 2001)
15. J. Navarro, S. Iyer, P. Druschel, A. Cox, Practical, transparent operating system support for Superpages, in *Fifth Symposium on Operating Systems Design and Implementation (OSDI'02)*, Boston, MA, December 2002
16. D. Raymond, D. Wood, Grail: A C++ library for automata and expressions. J. Symb. Comput. **11**, 341–350 (1995)
17. J. Rushby, A trusted computing base for embedded systems, in *Proceedings 7th DoD/NBS Computer Security Conference*, September 1984, pp. 294–311
18. J. Saltzer, Protection and the control of information sharing in Multics. Commun. ACM **17**(7), 388–402 (1974)
19. F.B. Schneider, Enforceable security policies. ACM Trans. Inform. Syst. Secur. **3**(1), 30–50 (2000)
20. G.W. Smith, R.B. Newton, A taxonomy of organisational security policies, in *Proceedings of the 23rd National Information Systems Security Conference*, Baltimore, MD, USA, October 2000
21. D.F. Sterne, On the buzzword "security policy", in *Proceedings of the 1991 IEEE Symposium on Security and Privacy*, Oakland, CA, 1991, pp. 219–230
22. E. Witchel, J. Cates, K. Asanovic, Mondrian memory protection, in *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, San Jose, CA, October 2002
23. M.E. Zurko, R.T. Simon, User-centered security, in *Proceedings of the 1996 Workshop on New Security Paradigms*, Lake Arrowhead, CA, September 1996

Chapter 6
Spatial Separation with Moats

Abstract This chapter describes moats and drawbridges (Huffmire et al., Proceedings of the 2007 IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 2007), a method for separating multiple cores on a single reconfigurable chip. Moats provide logical isolation by placing cores into distinct areas of the chip in a verifiable manner. Drawbridges use interconnect tracing to statically verify that only legal connections between system elements are allowed and that interfaces carrying sensitive data have not been tapped or routed to other cores or I/O pads. To facilitate legal communication between cores, two alternative communication architectures are compared.

6.1 Overview

Consider an FPGA system consisting of two processor cores and a shared AES encryption core, all on the same FPGA device. Further details about this system can be found in Chap. 7. Each of three cores requires access to off-chip memory to store and retrieve data. How is it possible to ensure that one processor's encryption key cannot be stolen by the other processor by reading the key from external memory or directly from the encryption core? These systems lack virtual memory, and the circuit produced by the CAD tool is an entangled labyrinth of gates and wires. To prevent theft of the key from the encryption core, a technique is needed to isolate the encryption engine from the other cores at the gate level. Protecting the key in external memory requires implementing a memory protection module as described in Chap. 5. In addition, it is also necessary to ensure that every memory access goes through the reference monitor and that the reference monitor is isolated and protected. It is also necessary to ensure that cores may only communicate through specified interfaces to prevent unauthorized access to the key in the encryption core and to prevent snooping on interconnections. A minor change to one of the last stages of the design flow constrains the placement of system elements. The techniques presented in this chapter are steps towards a cohesive reconfigurable system design methodology that facilitates the composition of systems using cores with different trust levels on a single chip.

T. Huffmire et al., *Handbook of FPGA Design Security*,
DOI 10.1007/978-90-481-9157-4_6, © Springer Science+Business Media B.V. 2010

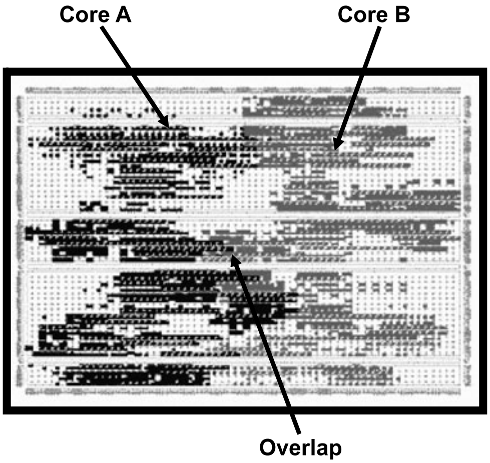


Fig. 6.1 A simple two-core system mapped onto a small FPGA. Each *gray* or *black square* represents an individual switchbox with high wiring complexity. Constraints must be imposed on the design tools to prevent cores from overlapping, which would increase the risk of unintended information flows. These constraints are also needed to make the task of statically analyzing a large design computationally feasible

6.2 Separation

Separation is a fundamental computer security concept that combines the ideas of isolation and controlled sharing. Cryptographic systems such as encryption devices were among the first to necessitate the development of strong isolation, since classified plaintext that is carried over red wires must be segregated from ciphertext carried over black wires. Saltzer and Schroeder define complete isolation as a "protection system that separates principals into compartments between which no flow of information or control is possible" [7]. Since the functionality of systems is greatly reduced if all of their components are totally isolated, a technique is needed to facilitate the controlled sharing of data among components. A solution for achieving controlled sharing in FPGA systems employs moats to provide the isolation and drawbridges to provide a means of precise sharing.

6.3 Physical Isolation with Moats

Since synthesis tools optimize a design's layout using performance as an objective function, the resulting circuit's logical elements and interconnections are entangled, as shown in Fig. 6.1. Each square represents a single switchbox, the associated LUTs, and associated routing. For an FPGA with 30K switchboxes, the task of static analysis of the bitstream is computationally infeasible. To protect a core's data and to prevent interference with a core's operation, the following section describes the use of moats to ensure isolation of the cores.

6.4 Constructing Moats

By imposing minor constraints on the design tools, moats are able to spatially isolate cores to enhance security. Design tools such as Xilinx's PlanAhead facilitate this

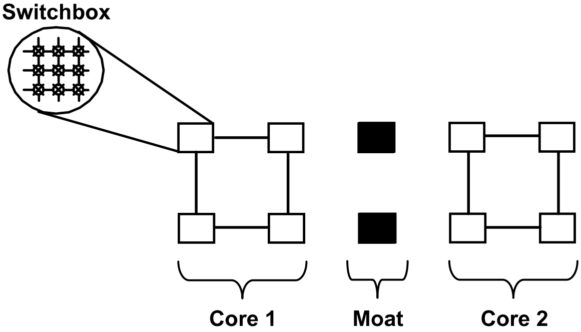


Fig. 6.2 The gap method surrounds each core with a *dead* area. This example restricts the fabric to only use routing segments that can span one switchbox. Under this constraint, the moat must have a width of at least one. Since the switchboxes shown in black are disabled, single length connections cannot be used to go from Core 1 to Core 2. However, a routing segment of length two could bridge the moat

process by giving designers fine grain control over the placement of cores. This section describes two methods for constructing moats, both of which place the cores in physically distinct regions:

- The *gap* method involves surrounding each core with a *dead* area (i.e., a moat). Then, if only routing segments that are shorter than the size of the moat are used in the design, the area inside the moat is completely isolated.
- The *inspection* method methodically checks the routing segments close to a core's border to ensure that no segments cross into the interior of the core, making it possible to reduce or eliminate the dead zone and to loosen the restrictions on the use of routing segments. Segments near a core's border are subject to greater restrictions than segments that are far from the edge.

For both methods, a core can only communicate with the world outside the moat via a precisely defined path called a *drawbridge*. While the gap method uses spatial isolation, the inspection method uses logical isolation.

6.4.1 The Gap Method

The gap method surrounds each core with a physical moat, which is a region in which the switchboxes have been disabled except for precisely defined paths for inter-core communication (drawbridges, discussed later). The Xilinx Virtex family of FPGAs uses routing segments that span either 1, 2, or 6 Configuration Logic Blocks (CLBs). Longline segments span an entire row or column. The ability to bypass intermediate switchboxes greatly improves performance because each switchbox introduces delay. The gap method sacrifies the performance gains of using longer segments in order to achieve spatial isolation.

Figure 6.2 shows the gap technique of constructing moats. To extend this example, if the design tools are constrained to only use routing segments of length one and two, the moat must have a minimum size of two to prevent signals from crossing the moat. In general, a moat needs to have a width of at least w if the design is restricted to never use routing segments longer than w. Static verification of these properties is straightforward. If all of the routing transistors in a switchbox are set to be disconnected, the switchbox belongs to a moat. It is also necessary to verify that all switches connecting to segments longer than w are off. Performing this verification requires some information about the bitstream, including the location of each switch's configuration bit. Since the JBits API can provide this information, these methods are applicable to any architecture supported by the JBits API [3], and extending JBits to other architectures is straightforward for vendors who are willing to do so.

6.4.2 The Inspection Method

The use of methodical checking makes it possible to reduce or even eliminate the moat. A *seamless* moat is one for which there is no gap at all (i.e., no disabled switchboxes). The key idea is to allow the use of segments longer than w if they are sufficiently far from the border. For example, even if w is 2, a hex segment could be used in the middle of a core that is 20×20 square. Clearly, it is very important to use static analysis to make sure that segments do not cross the border, but only a *subset* of the connections within a core must be checked. Progressively less checking is required further from the border towards the center of the core, which makes the job easier, as shown in Fig. 6.3. For example, for a seamless moat, checking segments of length one is unnecessary at least one CLB from the border. Segments of length two need not be checked at least two CLBs from the border. Checking longlines is always necessary because they span the entire row or column.

The depth d in CLBs to which a segment must be searched depends on the segment length l (1, 2, or 6) and the moat width w:

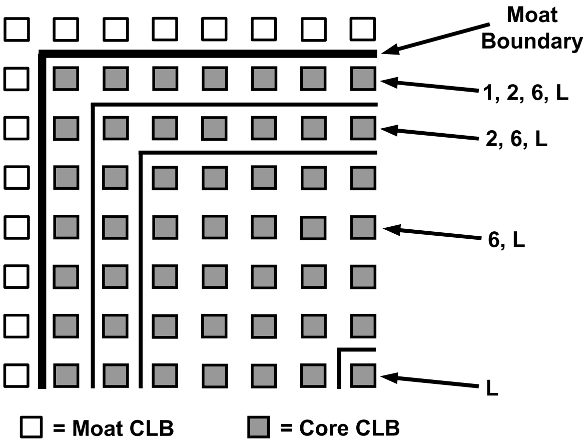
$$d(l,w) = \begin{cases} 0 & \text{if } l < w, \\ l - w & \text{if } l \geq w. \end{cases} \tag{6.1}$$

This equation demonstrates that if w is 1, there is no need to check segments of length one inside the core since such a segment cannot cross the moat. Even a segment on the border must utilize another connection inside the moat to escape, which will be detected during verification.

6.4.3 Comparing the Gap and Inspection Methods

For the gap method, analysis of the trade-off between circuit performance and moat area finds that small moats result in worse performance than large moats [5]. Re-

Fig. 6.3 The inspection method of moat construction allows for a smaller gap or no gap at all, called a *seamless* moat. Design tools must perform static checking for segments that cross the border. Progressively less checking is needed proceeding from the border towards the center of the core



stricting a design to use only short segments forces connections to pass through more switchboxes, each contributing some delay. It also requires more demand for switchbox resources, increasing circuit area. The first experiment restricts several benchmark circuits to use only segments of length one. The second experiment then compares this to the result of restricting the same circuits to use only segments of length one and two. A third experiment restricts the circuits to use only segments of length one, two, and six (i.e., no longlines). The results of these three experiments are then compared to the unconstrained *baseline*, which may use all segment lengths.

After using VPR to place and route the circuits, the area and critical path performance are measured. Although eliminating longlines has little effect, eliminating hex lines hurts area and performance significantly, and eliminating segments of length two furthers the degradation. To better understand the relationship of useful logic, inflation, and dead space, and their impact on area, an *effective utilization* metric is used. Effective utilization is simply the ratio of unrestricted logic to the sum of restricted logic and moat area. Assuming uniform rectangular cores, a moat size of two has the highest effective utilization, unless the number of cores is very large ($>$100) or very small (1). There is no need for a moat if there is only one core.

Analysis of the inspection method finds that small moats have better performance than large moats because the restriction on using longer routing segments is relaxed. An experiment applied the inspection method to three systems, including a dual-processor system-on-chip [6], a Multiple Input Multiple Output (MIMO) transceiver, and a JPEG encoder. For each system, the area and performance were compared for moat sizes of six, two, one, zero, and a baseline of no moat. As expected, systems with smaller moats use much less area but require more checking. However, the additional overhead of static checking is a small one-time-only cost. The impact on performance of moats constructed using the inspection method is minimal. Although seamless moats use the least area, using a nonzero moat size is often beneficial because the moat area can serve as a communication channel for the routing of drawbridge traffic. The next section discusses drawbridges, which are precisely defined paths for signals between cores and I/O pins.

6.5 Secure Interconnect with Drawbridges

Although moats isolate the cores, the cores must be able to communicate with each other in a controlled way. A *drawbridge* provides a precisely defined path for communication from one core to another core or I/O pad. It is necessary to specify in advance the location of the cores and the connections to be allowed. The drawbridge technique is applicable to multiple interconnection architectures, including direct connections and shared bus. Future work will extend drawbridges to networks-on-chip [2].

6.5.1 Drawbridges for Direct Connections

The system designer must first specify the legal connections so that the design can be statically analyzed for adherence to the specification. The route tracing tool described in [5] operates on the bitstream and a file that specifies the modules and interconnects, and it does not require design details of the cores, such as HDL, which may be proprietary and therefore unavailable. Checking at the very end of the design flow helps find illegal connections introduced in an earlier stage.

Moats allow the precise specification of the location of each core as well as the valid connections. The specification is simply a text file that defines all cores and I/O pins, including their location and a list of legal connections. Interconnect tracing involves bitstream analysis to determine the status of the switchboxes in order to trace the path along which a connection is routed. This mitigates the risk of illegal connections in the design, which will be detected by the tracing procedure.

The route tracing tool takes two inputs: a bitstream file and the specification file described above. Connections are specified in terms of a source (pin or module) and destination (pin or module). The tracing algorithm starts with a list of input and output pins (some pins may be able to do both) that can enter or leave a CLB. After performing a trace on all the input pins, it next traces all outgoing connections from all the CLBs in the modules. Finally, it performs a reverse trace on all outgoing connections from the modules. If the design is correct, this final step will not find any connections because the previous steps should have found them. The route tracing algorithm is a simple breadth first search with a few modifications: it maintains a list of every pin it has visited to prevent searching the same path twice. The search is terminated once another module is reached. The tracing program outputs all the connections it finds, and it can optionally display the route tree showing the entire path of a connection. When finished, it outputs whether or not the design was successfully verified. The following pseudocode describes the tracing process:

```
RouteTree trace(pin, module) {
   add pin to routeTree
   for all sinks of wire this pin is on {
      if sink is connected to pin
         if sink has already been searched
            return
         if sink is in another module
            check if connection is valid
            return
         add sink to list of searched pins
         trace(sink, module)
   }
}
```

Route Tracing Tool Input File Format. The first line of the input file to the route tracing tool starts with the letter *D* and specifies the device type:

```
D XC2V6000 FF1517
```

The next line of the file begins with the letter *N* and specifies the number of modules, pins, and connections:

```
N 4 5 12
```

The next lines of the file begin with the letter *M* and specify the module names as well as their xmin, xmax, ymin, and ymax coordinates:

```
M MB1 11 35 57 80
M MB2 11 35 13 35
M MB3 54 78 57 80
M MB4 54 78 13 35
```

The next lines of the file begin with the letter *P* and specify the pin names and whether they are input, output, or reset pins:

```
P B25 rst #Reset
P C36 in #rs_232_rx_pin
P J30 out #rs_232_tx_pin
P C8 in #rs_232_rx2_pin
P C9 out #rs_232_tx2_pin
```

The next lines of the file begin with the letter *C* and specify the connections: source, destination, and width:

```
C B25 MB1 1
C C36 MB1 1
C MB1 J30 1
C B25 MB2 1
C MB1 MB2 32
C MB2 MB1 32
C B25 MB3 1
C MB3 C9 1
C C8 MB3 1
C B25 MB4 1
C MB4 MB3 32
C MB3 MB4 32
```

Figure 6.4 shows the route tracing process. The black line shows a route connecting two parts of the design. The route tracing tool follows this route progressively from its beginning at the upper left of the figure to its end at the bottom right of the figure.

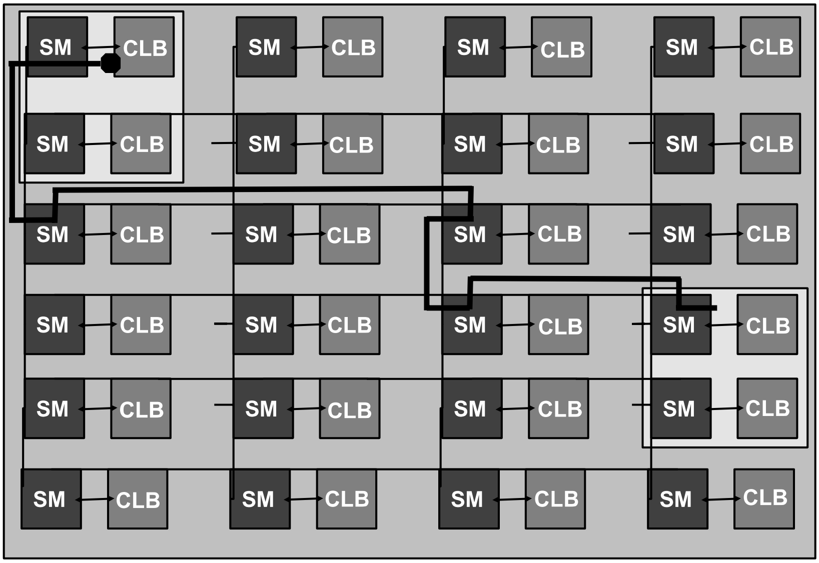


Fig. 6.4 Route tracing. *The black line* shows a route from *the light gray region* in *the upper left* consisting of two CLBs and two Switch Matrices (SMs) to *the light gray region* in *the lower right*. The route tracing tool follows this route progressively from beginning to end

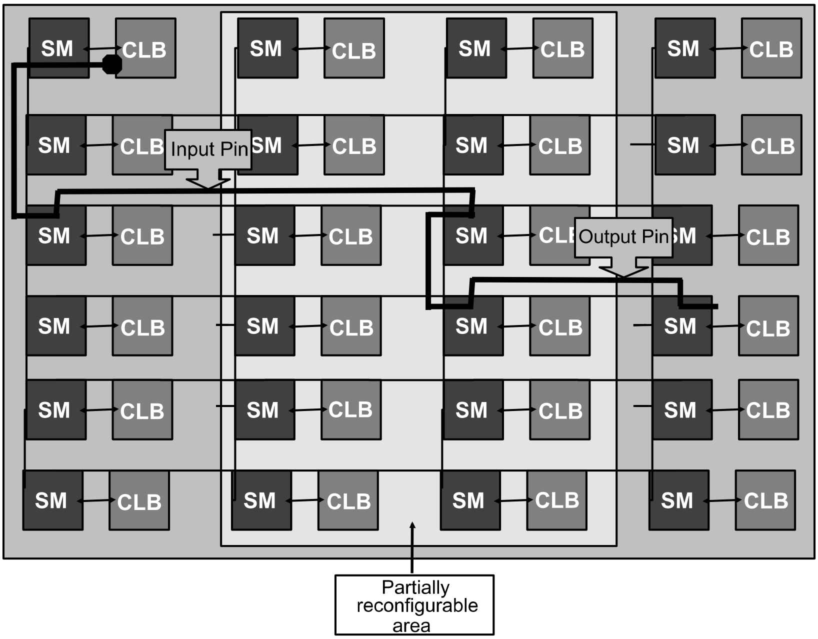


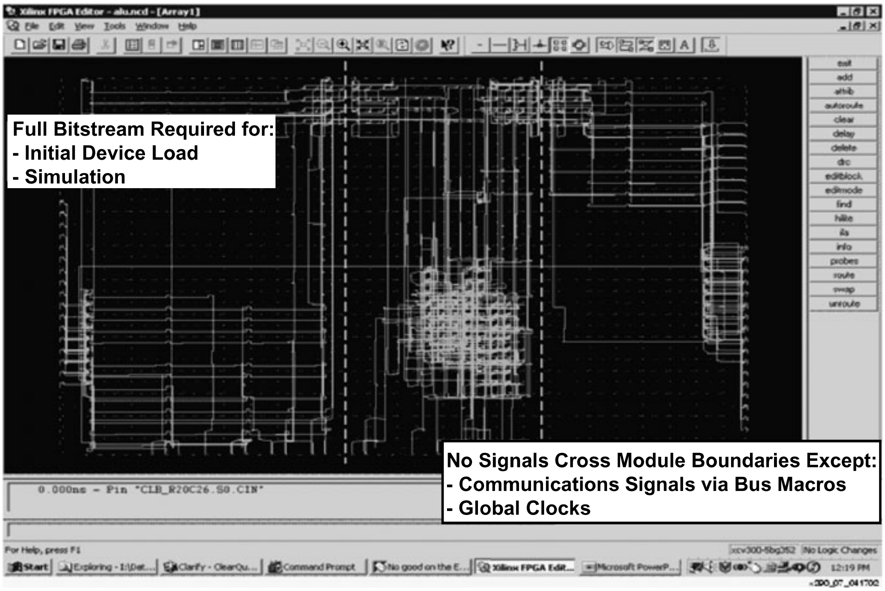
Fig. 6.5 Route tracing with partial reconfiguration. Only pins that enter or leave the reconfigurable portion of the design (*the middle region* shown in *light gray*) need to be searched, leading to a much faster tracing process

6.5.2 Route Tracing with Partial Reconfiguration

Figure 6.5 shows the route tracing process with partial reconfiguration. Partial reconfiguration makes the route tracing process much more efficient because it is only necessary to store two pins for each connection that passes through the reconfigurable portion of the design. Connections that enter but do not leave the area only require storing one pin and its direction (in or out). Only connections that enter or leave the reconfigurable portion of the design need to be searched. The cost of this efficiency gain is a small additional overhead of the initial tracing. Figure 6.6 shows the FPGA Editor view of a design that uses partial reconfiguration.

6.5.3 Drawbridges for Shared Bus Architectures

In addition to direct connections, cores can also communicate via a shared bus. However, the shared nature of a bus presents a security problem because a compromised core can snoop on the bus traffic. Even if all bus traffic is encrypted, the bus can be used to carry out a covert timing channel attack in which a *high* core sends



Full Bitstream Required for:
- Initial Device Load
- Simulation

No Signals Cross Module Boundaries Except:
- Communications Signals via Bus Macros
- Global Clocks

```
0.000ns - Pin "CLB_R20C26.S0.CIN"
```

Fig. 6.6 FPGA Editor view of a design that uses partial reconfiguration

data to a *low* core. The high core modulates its bus references, and the low core observes this modulation. To address these problems, an arbiter imposes time division multiplexing on the bus. Each core may only access the bus during its assigned time slice. Dividing the time equally among the cores eliminates covert timing channels on the bus, although it certainly limits the performance of the bus.

To mitigate the performance penalty of time division multiplexing, Hu proposes the idea of allowing a low core to *donate* time to a higher core [4]. The round robin begins with the lowest core and proceeds in order to the highest core. On the transition from the highest core to the lowest core, some housekeeping is needed, including using up donated time and clearing the cache.

An *ordered round robin* scheme can be used to provide even more flexibility than the donation scheme and can also provide the higher confidentiality processes the opportunity to use more of the CPU than their fixed time slice. In this approach, each equivalence class of processes is allocated a fixed, nominal time slice. Equivalence classes are placed in the circular scheduling queue according to the ordering of their labels, so that lower level processes execute before higher ones, except for the transition from highest to lowest. Figure 6.7 shows one round. Each class can use up to its nominal time slice and may give up the processor before then.

Additionally, with this *ordered round robin* scheme, after the lowest class, each equivalence class in turn has the opportunity to use up any slack time [1] donated by the lower level classes. When the highest equivalence class is scheduled, it is obligated to use up all of the donated time that remains for that round. Thus, the lowest class sees no variation in the amount of CPU time that higher classes use, and higher classes see only variations caused by lower classes. The intermediate

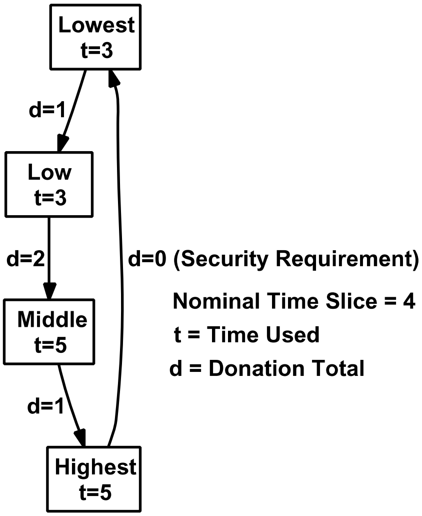


Fig. 6.7 Ordered round robin scheduling

classes can be regulated to ensure that the highest classes will have some donated time, if any exists, for example by restricting their use of donated time to a given percentage. If the percentage is incrementally increased for each of the higher levels, the result is somewhat akin to providing higher access classes with greater priority.

The *fixed* round-robin approach to secure MLS scheduling is to provide each equivalence class with a fixed time-slice in a round-robin scheduling scheme. However, this may waste resources when a process does not need all of its time slice. An *ordered priority* scheduling scheme can be used when the MLS labels are totally ordered, as long as the order of time slice assignments is inversely proportional to the MLS labels. This presents a problem in systems that need to provide higher confidentiality processes with higher priority. Additionally, if there are one or more non-hierarchical label components combined with a hierarchical label component (i.e., there is a partial ordering of labels), all processes with that hierarchical level must be assigned equal priorities (or equal time-slices in an ordered round-robin scheme)—an equivalence class.

In addition to covert channels, snooping must also be prevented. Placing an arbiter between each core and the bus limits each core's use of the bus to its time slice. Connecting the bus to memory requires the reference monitor technique described in Chap. 5. Each core can have its own arbiter (which requires a central timing multiplexer to handle scheduling), or a single arbiter can serve all the cores. Research [5] determined that a single arbiter is actually more efficient, and the arbiter can be integrated into the on-board peripheral bus (OPB) [6].

6.6 Protecting the Reference Monitor with Moats

The reference monitor described in Chap. 5 must be isolated, non-bypassable, and verifiable. To protect the reference monitor from tampering, it can be isolated with a moat. To prevent the reference monitor from being bypassed, drawbridges can be used to prevent a core from directly accessing the memory, snooping on the memory bus, or establishing an illegal connection with another core. Interconnect tracing ensures that the memory I/O blocks are only connected to the reference monitor.

References

1. A. Bavier, L. Peterson, D. Mosberger, BERT: a scheduler for best effort and realtime tasks. Princeton University Technical Report TR-602-99, Princeton, NJ, March 1999
2. S. Bourduas, Modeling, evaluation, and implementation of ring-based interconnects for network-on-chip. Ph.D. Dissertation, McGill University, Dept. of Electrical and Computer Engineering, Montreal, Canada, May 2008
3. S. Guccione, D. Levi, P. Sundararajan, JBits: Java-based interface for reconfigurable computing, in *Proceedings of the Second Annual Conference on Military and Aerospace Applications of Programmable Logic Devices and Technologies (MAPLD)*, Laurel, MD, USA
4. W.M. Hu, Lattice scheduling and covert channels, in *Proceedings of the 1992 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1992
5. T. Huffmire, B. Brotherton, G. Wang, T. Sherwood, R. Kastner, Moats and drawbridges: an isolation primitive for reconfigurable hardware based systems, in *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2007
6. T. Huffmire, B. Brotherton, N. Callegari, J. Valamehr, J. White, R. Kastner, T. Sherwood, Designing secure systems on reconfigurable hardware. ACM Trans. Des. Automat. Electron. Syst. (TODAES) **13**(3), 44 (2008)
7. J. Saltzer, M. Schroeder, The protection of information in computer systems. Proc. IEEE **63**(9), 1278–1308 (1975)

Chapter 7
Putting It All Together: A Design Example

Abstract This chapter describes a design example that incorporates the security primitives from the earlier chapters. This embedded system is connected to two separate networks that require encryption. It consists of two processor cores and a shared AES encryption core, all on the same device. Further details about a similar system can be found in Huffmire et al. (ACM Transact. Des. Automat. Electron. Syst. (TODAES) **13**(3):44, 2008).

A worked example of an embedded design is essential to understanding the practical issues of applying the security techniques described in the previous chapters. A real systems example also demonstrates how to design realistic policies. The example system is also useful for analyzing the usability of the security techniques as well as their impact on system performance and complexity. Specifically, this chapter describes the application of the reference monitor technique from Chap. 5, the moats and drawbridges technique from Chap. 6, and the covert channel analysis technique from Chap. 3 to a multi-core reconfigurable embedded system.

7.1 A Multi-Core Reconfigurable Embedded System

Figure 7.1 shows the design example, an embedded system connected to two separate networks, each labeled at a specific level, where data on both networks must be encrypted. The system consists of two MicroBlaze soft CPU cores, a shared AES crypto core, and two Ethernet interfaces, all running on a single FPGA device. Integrating multiple modules on a single device saves power, cost, and area. The system also includes off-chip DRAM. These components are organized into two compartments: a *gray* compartment consisting of one of the CPU cores and one of the Ethernet interfaces, and a *black* compartment consisting of the other CPU core and the other Ethernet interface. Each compartment needs to communicate with a network that is labeled at its own level. The components are connected to each other via

T. Huffmire et al., *Handbook of FPGA Design Security*,
DOI 10.1007/978-90-481-9157-4_7, © Springer Science+Business Media B.V. 2010

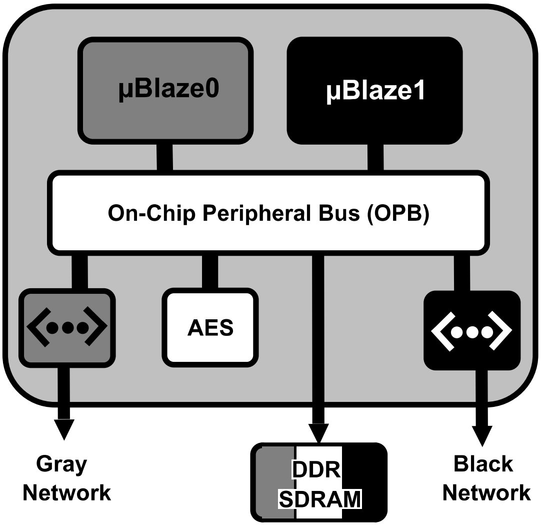


Fig. 7.1 A system consisting of two processors, a shared AES encryption core, two Ethernet interfaces, and shared external memory, all connected via a shared bus. Both a reference monitor and arbitration logic are integrated into this On-chip Peripheral Bus (OPB)

an On-chip Peripheral Bus (OPB), which contains arbitration logic and a reference monitor.

In order to avoid duplicating the crypto module (due to space constraints), it is shared between the gray and black compartments. Except for the AES engine, the two compartments must not share resources; one compartment must not be able to access the AES module when another compartment is using it, as that might allow information leakage between compartments; and the AES module must be purged between use by different domains. Effectively, the level of the AES module changes depending on who is using it: it is a single-level-at-a-time or *periods processing* device [6]. Periods processing is a term of art that refers to the process of *sanitizing* a shared resource after each classified job prior to its use by the next user. Sanitization is accomplished by destroying all memory residue of the prior job.

To satisfy these design requirements, moats spatially isolate all of the cores in this reconfigurable design, and the reference monitor enforces a temporal policy so that the two compartments can *periods process* the AES module in a manner that ensures the separation of the compartments. After each policy period, the AES engine is purged of data.

7.2 On-Chip Peripheral Bus

Since the job of a traditional shared bus is to connect multiple components together, not to separate them, it is necessary to modify the bus to facilitate controlled sharing between modules. The on-chip peripheral bus (OPB) is a soft IP core available in the Xilinx Embedded Development Kit (EDK) for connecting peripherals in an embedded system. The OPB was modified to incorporate a reference monitor that enforces a policy that logically separates the gray and black compartments. The two CPU cores are master devices, and the crypto core, Ethernet interfaces, and DRAM are slave devices.

7.3 AES core

A custom controller allows the AES crypto core to be controlled via shared memory. A processor requiring encryption or decryption places the plaintext to be encrypted or decrypted into a shared DDR SDRAM buffer in the processor's compartment. Then, the processor gives a signal to the AES core by writing to specific control words. This signal includes whether to encrypt or decrypt, the location of the data, and the size of the data. Upon completion of the transformation, the AES core writes the ciphertext or plaintext to the shared memory buffer and signals the processor via a control word. Finally, the initiating processor copies the transformed data from the shared buffer.

Since the AES core is effectively a shared device, access to it must be controlled by the reference monitor. Each buffer is located in the memory region of the initiating processor. This prevents processors from reading each other's plaintext. In addition, the reference monitor enforces a stateful *periods processing* policy [1, 5] that ensures that only one processor can use the AES core at a time.

7.4 Logical Isolation Compartments

Figure 7.1 shows the gray and black compartments of the embedded system. The gray compartment contains one of the MicroBlaze processors and one of the Ethernet interfaces. The black compartment contains the other MicroBlaze processor and the other Ethernet interface. The two compartments *periods process* the AES core, and the DRAM is partitioned among the compartments and the AES core. Separation of the gray and black compartments is essential because the gray and black networks operate at different levels. To achieve this separation, moats provide spatial isolation of cores, and the reference monitor provides temporal separation of system components.

7.5 Reference Monitor

In this embedded system, the reference monitor, which is integrated with the bus, mediates access to peripheral components, on-chip memory, and off-chip memory. The reference monitor either grants or denies memory access requests, and it can arbitrate access to any component connected to the OPB. An arbitration mechanism ensures that two cores cannot use the bus at the same time.

Using memory mapped I/O makes it possible to use the reference monitor to mediate access to the I/O devices and the shared memory, and it also makes it possible to scale up to larger numbers of master devices.

7.6 Stateful Policy

The formal top-level specification for the stateful policy is expressed using the low-level policy language described in Chap. 5. $Module_1$ corresponds to $\mu Blaze_0$, and $Module_2$ corresponds to $\mu Blaze_1$. The policy compiler translates this specification directly to a hardware description of a reference monitor that enforces the policy. The reference monitor is capable of enforcing a resource sharing policy because each component of the system besides the MicroBlaze processors is assigned a specific address range, expressed in the following productions:

$Range_1 \rightarrow$ [0x28000010, 0x28000777]; (AES_1)
$Range_2 \rightarrow$ [0x28000800, 0x28000fff]; (AES_2)
$Range_3 \rightarrow$ [0x24000000, 0x24777777]; ($DRAM_1$)
$Range_4 \rightarrow$ [0x24800000, 0x24ffffff]; ($DRAM_2$)
$Range_5 \rightarrow$ [0x40600000, 0x4060ffff]; ($Ethernet_1$)
$Range_6 \rightarrow$ [0x40c00000, 0x40c0ffff]; ($Ethernet_2$)
$Range_7 \rightarrow$ [0x28000004, 0x28000007]; ($Ctrl\_Word_1$)
$Range_8 \rightarrow$ [0x28000008, 0x2800000f]; ($Ctrl\_Word_2$)
$Range_9 \rightarrow$ [0x28000000, 0x28000003]; ($Ctrl\_Word_{AES}$)

AES_1 is the shared DDR SDRAM buffer in the gray compartment, and AES_2 is the shared DDR SDRAM buffer in the black compartment. One state of the policy ($Access_0$) corresponds to the case when neither $Module_1$ nor $Module_2$ is using the AES core. This is the initial state of the system. Another state ($Access_1$) corresponds to the case when $Module_1$ is using the AES core. A third state ($Access_2$) corresponds to the case when $Module_2$ is using the AES core. A processor gains access to the AES core by writing to $Ctrl\_Word_1$ ($Range_7$), which triggers a state transition ($Trigger_1$ for $Module_1$ or $Trigger_3$ for $Module_2$). A processor releases the AES core by writing to $Ctrl\_Word_2$ ($Range_8$), triggering another transition ($Trigger_2$ for $Module_1$ or $Trigger_4$ for $Module_2$). The following productions specify the three states $Access_0$, $Access_1$, and $Access_2$:

$$\begin{aligned}Access_0 &\rightarrow \{Module_1,\ rw,\ Range_5\}|\{Module_2,\ rw,\ Range_6\}\\ &\quad|\{Module_1,\ rw,\ Range_3\}|\{Module_2,\ rw,\ Range_4\}\\ Access_1 &\rightarrow Access_0|\{Module_1,\ rw,\ Range_1\}|\{Module_1,\ rw,\ Range_9\};\\ Access_2 &\rightarrow Access_0|\{Module_2,\ rw,\ Range_2\}|\{Module_2,\ rw,\ Range_9\};\end{aligned}$$

The following productions specify the transitions between the states, $Trigger_1$, $Trigger_2$, $Trigger_3$, and $Trigger_4$:

$$\begin{aligned}Trigger_1 &\rightarrow \{Module_1,\ w,\ Range_7\};\\ Trigger_2 &\rightarrow \{Module_1,\ w,\ Range_8\};\end{aligned}$$

$Trigger_3 \rightarrow \{Module_2,\ w,\ Range_7\};$
$Trigger_4 \rightarrow \{Module_2,\ w,\ Range_8\};$

The following productions use some rather complicated regular expressions to specify the structure of the state machine that enforces the policy:

$$\begin{aligned}
&Expr_1 \rightarrow Access_0|Trigger_3Access_2{}^*Trigger_4;\\
&Expr_2 \rightarrow Access_1|Trigger_2Expr_1{}^*Trigger_1;\\
&Expr_3 \rightarrow Expr_1{}^*Trigger_1Expr_2{}^*;\\
&Policy \rightarrow Expr_1{}^*|Expr_1{}^*Trigger_3Access_2{}^*\\
&\qquad\quad |Expr_3Trigger_2Expr_1{}^*Trigger_3Access_2{}^*\\
&\qquad\quad |Expr_3Trigger_2Expr_1{}^*|Expr_3|\varepsilon;
\end{aligned}$$

Building this policy demonstrated a limitation of the low-level language. In this case, due to the complexity of the regular expressions, it is easier to specify the state machine and generate the regular expressions using Grail [4], which can convert state machines to regular expressions using the *fmtore* (finite state machine to regular expression) program. The first step is to specify the basic structure of the DFA using the Grail language in a file called *grail_machine*:

```
(START) |- 0
0 A 0
1 B 1
2 C 2
0 D 1
1 E 0
0 F 2
2 G 0
0 -| (FINAL)
1 -| (FINAL)
2 -| (FINAL)
```

Figure 7.2 shows this DFA. Next, the *fmtore* program generates the regular expression from the finite state machine:

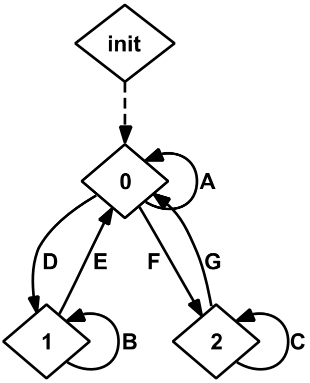
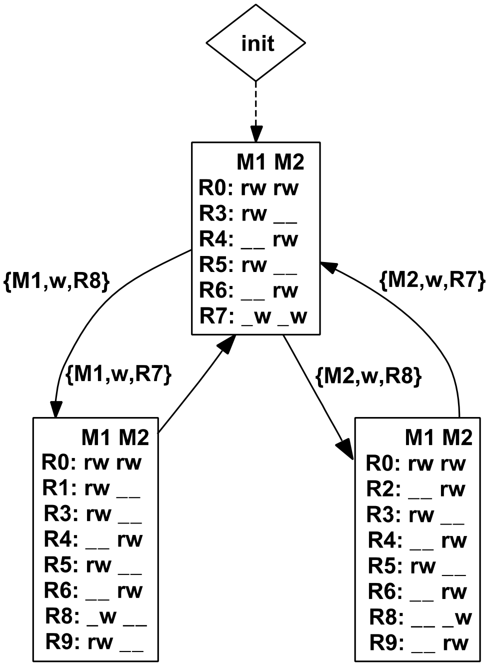


Fig. 7.2 In this case, a little help from Grail is needed to generate the regular expression

Fig. 7.3 The DFA that enforces the stateful policy for the embedded system



```
% ./fmtore grail_machine
(A+FC*G)*
+(A+FC*G)*FC*
+((A+FC*G)*D(B+E(A+FC*G)*D)*)E(A+FC*G)*FC*
+((A+FC*G)*D(B+E(A+FC*G)*D)*)E(A+FC*G)*
+((A+FC*G)*D(B+E(A+FC*G)*D)*)
+""
```

Note that A corresponds to $Access_0$, B corresponds to $Access_1$, C corresponds to $Access_2$, D corresponds to $Trigger_1$, E corresponds to $Trigger_2$, F corresponds to $Trigger_3$, and G corresponds to $Trigger_4$.

Solutions to this problem include extending the higher-level language described in Chap. 5 to handle a wider variety of stateful policies or providing the engineer with a tool that has a user interface for constructing policies, which are then generated automatically.

Figure 7.3 shows the DFA for the stateful policy. The design flow automatically generates a hardware description of a reference monitor from the policy specification. In the initial state of the system, neither core may use the AES core. This is the state pointed to by *init*, and the access matrix for $Access_0$ is shown. $Module_1$ obtains access to the AES core by writing to *control_word*$_1$, causing the transition to the state in the lower left, which shows the access matrix for $Access_1$. Since $Module_1$ can access *control_word*$_{AES}$ and *AES*1, it can copy plaintext or ciphertext into its portion of the AES core's memory (*AES*1) and signal the AES core appropriately via *control_word*$_{AES}$. Upon completion of the encryption or decryption, the AES core signals *Module*$_1$ via *control_word*$_{AES}$, and *Module*$_1$ copies the transformed data from the shared buffer. Then, *Module*$_1$ releases the AES core by writing to *control_word*$_2$, triggering a transition back to the initial state. *Module*$_2$ obtains and releases access to the AES core in a similar fashion. The reference monitor prevents both *Module*$_1$ and *Module*$_2$ from using the AES core at the same time. This scheme encodes the policy rules into the hardware so that the hardware enforces the policy by design.

7.7 Secure Interconnect Scalability

Systems with large numbers of cores will require a more sophisticated strategy for managing the communication between the cores in an efficient and secure manner than a single bus with a single reference monitor. Depending on the choice of interconnect, distributed reference monitors may be needed. For some stateful policies, synchronization of the state of multiple reference monitors can be a challenge. A security architecture can minimize the synchronization overhead by reducing the number of reference monitors required. For example, cores can be organized into spatially isolated equivalence classes, and cores within a group only communicate with each other via local interconnect. The reference monitor only needs to mediate communication that crosses the boundaries between equivalence classes.

Future embedded systems-on-a-chip will use advanced interconnect technology, including network-on-chip grids [2]. Managing security in these designs will require integrating enforcement mechanisms into the on-chip routers.

7.8 Covert Channels

In the example from Fig. 7.3, a covert channel analysis technique such as that from Chap. 3 provides analysis of possible covert channels from *Module*$_1$ to *Module*$_2$ (from the gray processor to the black processor). In other words, the gray processor could send information to the black processor by using the internal state of the reference monitor as a covert storage channel. Several covert channels are possible. One is based on interference in access to the bus. Another is based on the ability of the gray processor to modulate the system between policy *Access*$_1$ and *Access*$_0$, and for the black processor to detect those changes. The bandwidth of this covert channel depends on how frequently the gray processor grabs the AES core. One solution involves enforcing a time division multiple access (TDMA) sharing scheme. In other words, the gray compartment uses the AES core for a fixed amount of time, then the black compartment, and so on. The system can also require that the AES core be used for a sufficiently long time that the bandwidth of the covert channel is low enough. The system can also measure the number of times the gray core grabs the AES core and raise an alarm if this activity exceeds a threshold. The system can also introduce noise into the covert channel by randomly varying system events.

To address the problem of the internal state of the AES core itself from being used to send information from the gray core to the black core, an object reuse mechanism is needed. A very heavy-handed approach is to use dynamic partial reconfiguration to erase the entire contents of the moat-bounded region in which the AES core resides and then reload a fresh AES core's configuration into that space. A smarter object reuse technique would only erase the stateful elements of the AES core, but this finer-grained approach requires rigorous analysis. How do you know that you erased everything and that nothing was left behind?

7.9 Incorporating Moats and Drawbridges

Incorporating moats and drawbridges into the design provides spatial separation and simplifies the job of verification. Moats protect the reference monitor by providing spatial isolation of the system components, including the reference monitor itself. Proper configuration of drawbridges prevents the reference monitor from being bypassed by detecting any illegal connections between system components. Specifically, the tracing algorithm detects connections from cores and memory that bypass the reference monitor. It also detects illegal connections between cores that bypass the reference monitor and illegal connections that allow cores to snoop on each other's memory traffic.

Xilinx PlanAhead [7] is used to construct the moats. As shown in Figs. 7.4 and 7.5, PlanAhead provides a visual interface for placing cores on the chip, and the designer can leave a gap between cores. The output of PlanAhead is a user constraints file that is used in the synthesis step. The synthesis tool calculates the optimal layout for each core within its confined region.

The design is separated into seven distinct regions, one for each of the components. Determining the optimal placement of the cores, which significantly impacts

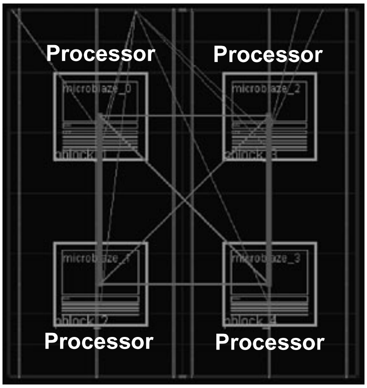


Fig. 7.4 Constructing moats with the PlanAhead tool for an example design with four MicroBlaze processor cores

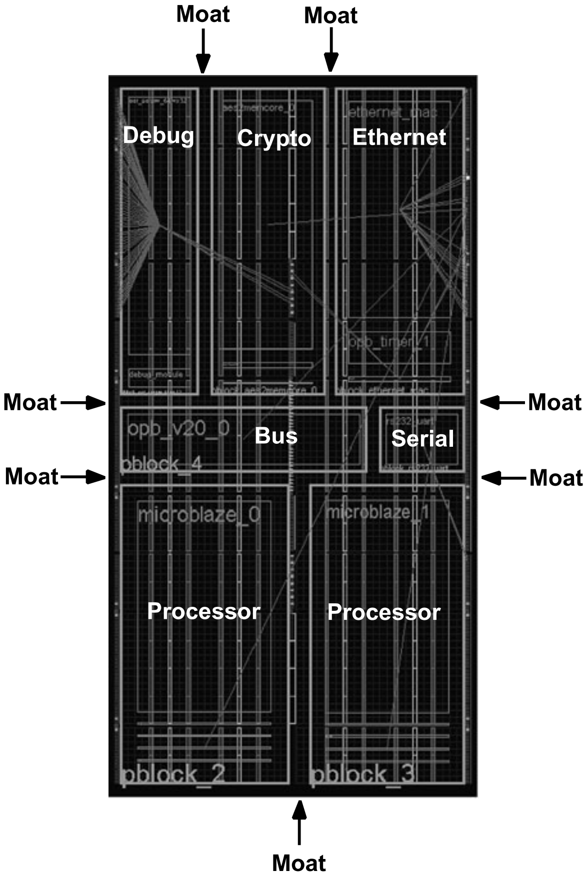


Fig. 7.5 Layout of the system in PlanAhead after partitioning into seven moated cores

system performance, requires some trial and error. Cores that need to communicate with each other should be placed close together, and other cores should also be placed close to the I/O pins. Multipass place and route is used to compare various layouts.

7.10 Implementation and Evaluation

The Xilinx Platform Studio (XPS) software is used to assemble the embedded design, and Modelsim is used to test the cores and the custom OPB on a set of test inputs. The policy complier generates the Verilog description of the reference monitor. XPS is also used to develop the software that runs on the FPGA, driving the MicroBlaze processors. The Xilinx Microprocessor Debugger (XMD) is used to debug this software. Using the inspection method of constructing seamless moats, experiments find negligible impact on performance and no impact on area [3].

7.11 Software Interface

Continuing with the gray/black example, a programmatic interface allows an application running on a PC or laptop to send data and keys to the device and to receive the resulting ciphertext or plaintext programmatically. The user interface, implemented in C++, specifies the operation (encryption or decryption), the input data file, the key file, and where to save the result. In a typical configuration, a *gray* laptop connects to the FPGA board via one of the Ethernet interfaces, and a *black* laptop connects to the FPGA board via the other Ethernet interface.

7.12 Security Usability

Security techniques are useless unless designers can easily make use of them. The experience of building this embedded system shows that moats are a simple yet effective type of floor planning and are straightforward to implement. Determining the floor plan with the optimal performance requires some trail and error, but this is also straightforward for experienced designers. Incorporating the reference monitor is facilitated by the OPB framework provided by Xilinx. The policy compiler also facilitates the integration of the reference monitor because modifying the policy merely requires recompiling. However, experience also shows that a custom tool such as we describe is needed for constructing policies without requiring that the designer be a regular expressions guru.

7.13 More Example Security Architectures

In a system that conforms to a Bell and LaPadula information flow policy, the computational resources are partially ordered. For example, information can flow from a C core to a S or TS core, but not from a C core to a U core. If there are many cores and many direct connections, the number of rules in the policy implemented as a table will become large, which will make the reference monitor large. Making many copies of the reference monitor for each direct connection will also waste resources. Instead of a reference monitor for every direct connection, something akin to a *diode*, which only allows information flow in one direction, can be placed on direct connections. Using diodes between compartments that have been isolated with moats greatly reduces the arbitration/reference monitor resources needed to mediate communication. This simplicity comes at the cost of less granularity of enforcement.

7.13.1 Classes of Designs

The design space of security architectures for FPGA systems can be divided into two classes: Monolithic and Cores + Memory.

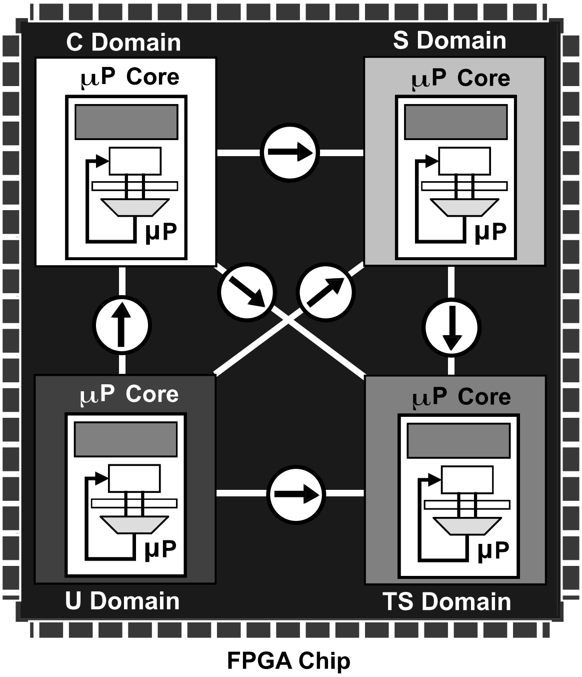


Fig. 7.6 This system has four security compartments that are isolated using moats. Each compartment contains a CPU core. The four compartments are arranged in a *mesh* configuration, and *diodes* ensure the proper flow of information between compartments. Data can flow from the U compartment to the C compartment; from U to S; from U to TS; from C to S; from C to TS; and from S to TS. This design class does not take memory into account

7.13.1.1 Monolithic

This design class is only concerned about monolithic computing resources and does not consider memory. Figure 7.6 shows an example of this class.

7.13.1.2 Cores + Memory

This design class takes memory into account, and memory is partitioned into ranges. Each range is assigned a security label, and the same set of labels is also used for the cores, which establishes *compartments* or *partitions*. Since memory can reside either on the chip or off the chip, this design class can be further divided into those security architectures that consider on-chip memory, off-chip memory, or both. An on-chip reconfigurable reference monitor provides memory protection for both the on-chip SRAM and off-chip DRAM. Moats provide protection for the on-chip Block RAM (BRAM). Figure 7.7 shows an example of this design class.

The reference monitor enforces a policy that specifies which cores may access which memory ranges. The DRAM is partitioned into ranges, and each range is assigned a security label. The same set of labels is used for both the cores and the ranges. To conform to a Bell and LaPadula information flow policy, the reference monitor must not allow a core to write to a range with a lower label or read a range with a higher label.

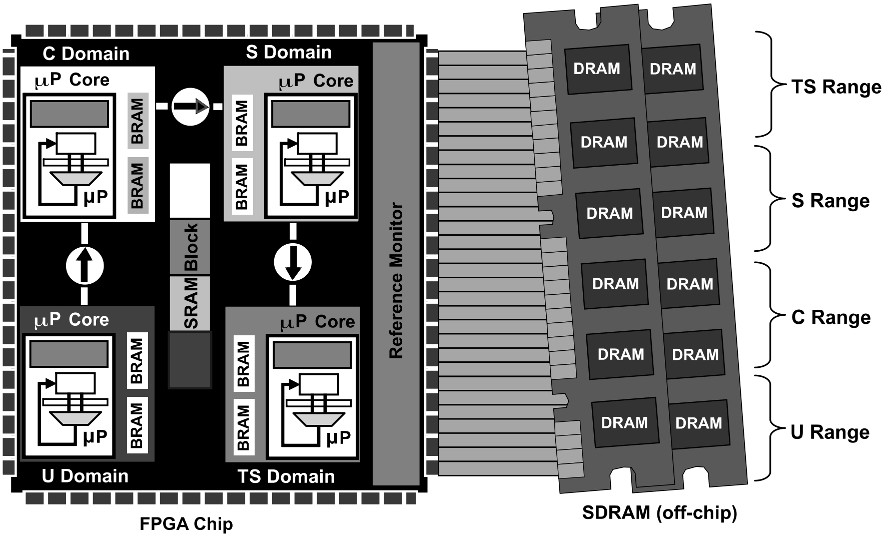


Fig. 7.7 This system is a representative of a class of security architectures that takes both on-chip and off-chip memory into account. Each compartment contains a CPU core and two Block RAM (BRAM) blocks that are isolated using a moat. A reference monitor provides memory protection for both on-chip SRAM and off-chip DRAM

7.13.2 Topologies

Topologies are different arrangements of cores. Figure 7.8 shows a variety of communication topologies arranged in a hierarchical fashion. Dividing the computing resources into compartments in an intelligent manner can greatly reduce the complexity and area of the policy enforcement mechanisms. Clever placement of the reference monitor and the use of diodes also can reduce the complexity of the mechanism that enforces the intended policy. As advances are made in silicon fabrication, more and more cores can fit on a chip. Since the number of possible direct connections between cores grows as the square of the number of cores, managing all this communication in an efficient and secure manner becomes very important.

Diodes are an ideal enforcement tool for some direct connections because directional connections are very simple to build in hardware and are much smaller than a reference monitor. All that is needed is a way to check that diodes are placed where they are needed and that they point in the right direction, according to the policy. However, there may be cases where a reference monitor is preferable to a diode, such as a stateful policy in which it is necessary to temporarily disallow a particular communication link.

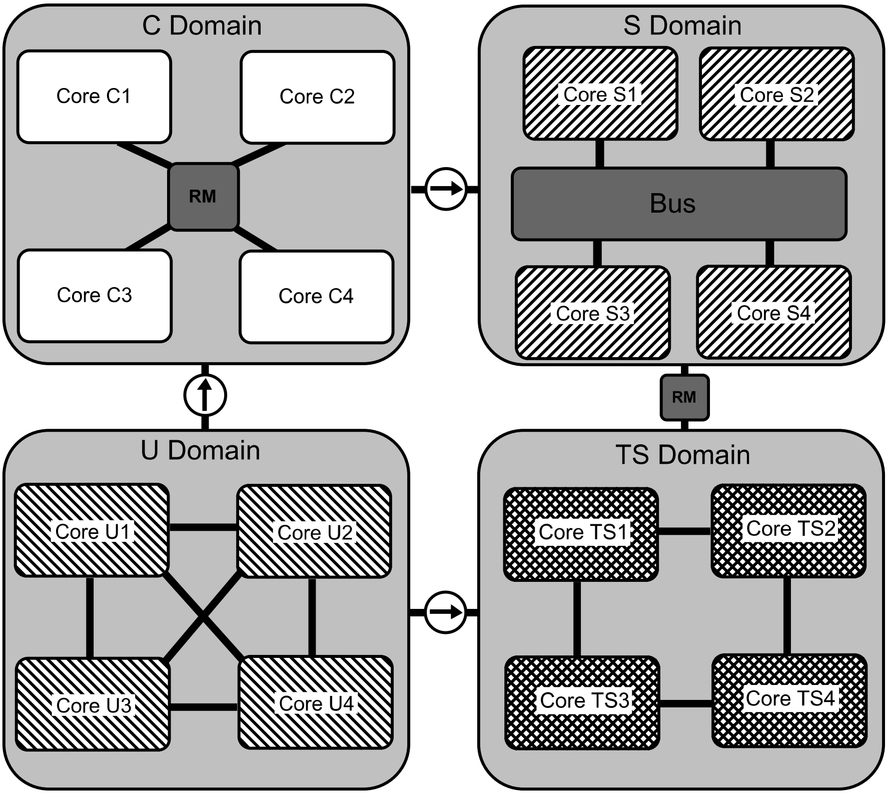


Fig. 7.8 This figure shows a variety of communication topologies arranged in a hierarchical fashion. The TS compartment has four cores arranged in a grid. The S compartment has four cores connected via a bus. The C compartment has four cores arranged in a *star* configuration, with a reference monitor that accounts for least privilege. The U compartment has four cores arranged in a mesh configuration. Diodes ensure that data can only flow from U to TS; from U to C; and from C to S. A reference monitor is located on the connection between TS and S in order to enforce a more nuanced policy (e.g., flow from S to TS only under certain circumstances)

7.14 Summary

Security must become a first-class design constraint for embedded systems, which are often incorrectly assumed to be secure. This design example demonstrates the application of the security primitives described in this book. The experience of building this system also shows that a custom utility is useful to facilitate the construction of stateful policies involving complicated regular expressions. There are many possibilities for future work, including integrating the reference monitor into the direct memory access (DMA) controller in an efficient manner and designing policies for systems that use a DMA controller. The problem of denial-of-service is left to future work, specifically the problem of a malicious core making repeated illegal requests.

References

1. J.P. Anderson, Computer security technology planning study. Technical Report ESD-TR-73-51, ESD/AFSC, Hanscorn AFB, Bedford, MA, 1972
2. S. Bourduas, Modeling, evaluation, and implementation of ring-based interconnects for network-on-chip. Ph.D. dissertation, McGill University, Dept. of Electrical and Computer Engineering, Montreal, Canada, May 2008
3. T. Huffmire, B. Brotherton, N. Callegari, J. Valamehr, J. White, R. Kastner, T. Sherwood, Designing secure systems on reconfigurable hardware. ACM Transact. Des. Automat. Electron. Syst. (TODAES) **13**(3), 44 (2008)
4. D. Raymond, D. Wood, Grail: A C++ library for automata and expressions. J. Symb. Comput. **11**, 341–350 (1995)
5. US Department of Defense, National Industrial Security Program Operating Manual (NISPOM), 28 February 2006
6. C. Weissman, Secure computer operation with virtual machine partitioning, in *Proceedings of the National Computer Conference and Exposition*, Anaheim, CA, May 1975
7. Xilinx Inc., PlanAhead Methodology Guide, San Jose, CA, 2006

Chapter 8
Forward-Looking Problems

Abstract This chapter considers forward looking problems, including trustworthy tools, formal verification of hardware designs, configuration management, languages, physical attacks, design theft, and securing the entire manufacturing supply chain.

8.1 Trustworthy Tools

Creating hardware designs that are free of covert channels remains a difficult problem. Future research is needed to develop trustworthy design flows, and this will likely represent a large long-term research effort. For example, FPGA design tools are very large and complex, with many features such as optimization functionality. Tools also have a proprietary nature, and vendors are continually adding new features. A full installation of the FPGA design tools can consume tens of gigabytes of hard drive storage. However, it may be possible to design a much smaller design flow with an essential subset of functions. This design flow can be developed by a small group of people and subjected to rigorous formal security analysis to provide a greater level of trustworthiness than the full-blown version of the tools. In the case where synthesis optimizations are removed, performance concerns may require that only part of the design can undergo trustworthy compilation (e.g., security mechanisms). *However, the limited nature of this stripped-down design flow tool will not necessarily mean that the resulting security-critical hardware modules will be less efficient than those generated by the full commercial tool chain.*

It will not be easy to develop a stripped-down, trustworthy design flow. In the software world, Thompson's famous paper describes the difficulty of detecting subversion of a compiler [10]. In addition, analyzing an IP core or a computer program to determine whether it is malicious is not decidable in the general case, since such analysis is equivalent to the halting problem, according to Rice's theorem [7].

Another key aspect of trust is that the output of each stage of the hardware design flow faithfully implements the input to that stage. The application of existing compiler verification techniques should be investigated [3]. Complete physical isolation

T. Huffmire et al., *Handbook of FPGA Design Security*,
DOI 10.1007/978-90-481-9157-4_8, © Springer Science+Business Media B.V. 2010

of certain modules of the design assists formal verification because those modules that are not being tested can be *masked out* from the analysis. Assuming that the correct behavior of each isolated component is well-defined, each component can be tested separately to ensure that the output of the design flow is correct. In the case where modules interact during runtime, their composed behavior must be tested, in general. In addition to static analysis, runtime mechanisms should be developed to verify the correct operation of the circuit and to verify the integrity of the circuit.

8.2 Formal Verification of Secure Systems

Research is needed to determine how well-understood techniques for software security can be efficiently translated to the hardware realm [5]. Applying formal methods to the software design process involves using established techniques such as theorem proving and model checking to ensure that the software satisfies security requirements. Specifically, the code is checked against a *formal top level specification* (*FTLS*). Unfortunately, theorem proving and model checking are not a silver bullet because there may be a flaw in the model, the proof, the implementation of the specification, or the physical deployment environment. In addition, scalability can be a problem for model checking because of the growth in the complexity of the security analysis with the size of the code under analysis. Although theorem proving is more scalable than model checking in that regard, it is time-consuming and requires experienced people to develop models and to guide the theorem prover. While automatic theorem proving is an NP-complete problem, we look forward to advances utilizing techniques such as have been used in model checking to reduce extremely large yet partitionable search spaces.

Using the Common Criteria for system evaluation has many practical limitations. A common critique is that the Common Criteria simply evaluates the Target of Evaluation against the Protection Profile, but if the Protection Profile is flawed, the evaluation is not necessarily useful. Both formal methods and the Common Criteria can be difficult to adapt to the reality that systems are developed in an iterative manner. When the end product differs substantially from the system at the beginning of the project, a change to the specification can significantly impact both formal methods analysis and Common Criteria evaluation. Research is needed to reduce this problem.

Given the lack of tools and theory with which to construct trusted systems, designers should pay particular attention to the security-critical components and interfaces in any design. In a complex system with many requirements, no single security technique by itself may be able to provide adequate security. Multiple complementary mechanisms must work together, requiring a holistic view of the entire system. These include a coherent security architecture for runtime enforcement, static analysis, formal methods, cryptography, security usability, user training, and many others. Although perfect security does not exist in the real world, systems must be engineered as perfectly as possible so that the adversary must work hard to attack each individual machine, rather than letting multiple systems fall like dominoes after one of them is compromised [6].

8.3 Security Usability

Good security needs to consider users: end users, administrators, and developers. Developers must be able to easily adopt security techniques if they are to be incorporated into real designs. End users must be trained on the proper use, configuration, management, and update of systems, and they must be aware of the risk of social engineering attacks. Sound policies for users must be created and implemented, including intrusion detection and audit policies to address insider attacks. Security analysis should consider the assumptions regarding personnel and the environment. Development machines and tools should be controlled and protected, and trusted delivery of components from the factory should be adopted. Policies must be expressed in a clear, intuitive way so that they can be designed correctly by developers.

8.4 Hardware Trust

While the reconfigurable reference monitor technique is a first step in applying rigorous design and analysis to FPGA design security, more work is needed. Another opportunity for future work is the development of a secure form of virtual memory to provide resource arbitration for FPGAs. Research is also needed to ensure that the FPGA manufacturer has not inserted malicious circuitry into the FPGA fabric via a developmental insider attack. Although the attacker does not know what design will eventually be loaded onto the device, for highly trusted applications, even the remote possibility of such an attack succeeding is a serious concern. Ideally, an attacker attempting to insert a fault into the fabric should not be able to do any better than random chance (i.e., cause a random fault). The application of existing processor design verification techniques [4] to the FPGA fabric should be investigated.

8.5 Languages

Applying formal methods with respect to formally defined properties of hardware designs and hardware design flows would be another very useful long-term research project. For example, formal methods could be applied to Hardware Description Language (HDL) code such as Verilog or VHDL [2, 8]. Formal methods could also be used to verify that the output of an HDL compiler faithfully implements the input. Translation of a high-level specification to a low-level implementation remains an open research challenge.

Another opportunity for future work is to develop enhancements for HDLs that make it easier for designers to improve the security of their designs. For example, HDLs could incorporate the notion of *safety* with respect to information flow, as has been done for specialized programming languages. For example, the security label of a wire could be specified in HDL. These language enhancements could be used in conjunction with static analysis techniques that analyze the HDL for conformance with safety properties, or with runtime techniques that make use of the language enhancements using runtime mechanisms. In addition, configuration management techniques could be applied to HDL code as has already been applied to software code.

In Chap. 5, we describe the use of a customized language for expressing a formal top level specification (FTLS). The FTLS describes a policy that specifies the legal sharing of memory among cores on an FPGA. A compiler translates the policy directly to a hardware description of a circuit that enforces the policy. Applying formal methods to ensure the correctness of the HDL description of the circuit or that the circuit faithfully implements the HDL description is left to future work. The scope of such a project is enormous, since the design space of embedded systems that can be loaded onto an FPGA is infinite, and a proof only applies to one system. The scope must also consider the level of system abstraction at which a proof is constructed (e.g., chip level, board level, system level) as well as whether the proof is applied to the FPGA fabric itself, the bitstream encryption mechanisms, the bitstream containing the reconfigurable design, or the software running on top of the reconfigurable hardware. In addition, the policy language could be enhanced with features that make it impossible to compose a flawed FLTS that results in the covert storage channel described in Sect. 3.5.

8.6 Configuration Management

Even so, designers should apply configuration management techniques from the software discipline to the hardware discipline. For example, software configuration management (CM) establishes a repository of specific versions of tools (e.g., compilers) that have a good reputation. A specific version of a tool must earn its reputation in the design community. Prior to installing the tool, an engineer can verify the cryptographic checksum of the downloaded archive or disk image. In addition, the output of the design tools should be analyzed with respect to known test cases. Finally, just as software CM can be applied to specific versions of open-source software libraries (e.g., the GNU C Library), hardware CM should be applied to specific versions of open-source cores (e.g., an Ethernet core available at opencores.org).

8.7 Securing the Supply Chain

In addition to trusted design kits, tool flows, and design libraries, securing the entire semiconductor manufacturing supply chain will require trusted packaging, assembly, and delivery. To mitigate malicious hardware *viruses*, research is needed to understand, categorize, and analyze malicious hardware so that it can be detected. For example, how can we prevent a denial-of-service attack by a *malicious* core? Penetration testing and other forms of dynamic analysis, if used properly, can be useful for detecting specific vulnerabilities, although it certainly cannot detect all subversions. Although developers perform functional testing, to be most effective, both developers and external testers should perform penetration testing. Specific criteria are needed regarding who does the testing and the extent of testing to be performed. Both will be determined based on how critical the system is. Understanding the differences between penetration testing of software and hardware is also needed. Results from research on malicious software, including discoveries about the theoretical limits of detecting subversion in software [9], will also apply to hardware.

8.8 Physical Attacks on FPGAs

Preventing physical attacks on FPGAs will require analysis of specific models of FPGAs to identify vulnerabilities to physical attacks, including probing, power analysis, thermal channels (using variations in temperature to encode information), electromagnetic radiation analysis, timing analysis, and sand-and-scan attacks. Results of this analysis can be used to incorporate mitigations (e.g., masking) into the FPGA itself, the design loaded onto the FPGA, or both (e.g., tamper-sensing mesh and epoxy potting material). Side channel attacks can compromise not only the keys and data being processed by the FPGA but also the keys used to encrypt the bitstream, making it possible to thwart bitstream encryption schemes, which vary from vendor to vendor and from model to model. Since traditional masking techniques have theoretical limits according to results from information theory, total elimination of side channels is an open problem. Lowering the bandwidth of these channels is the best that current techniques can achieve. System designers should consider this fact during risk analysis.

A specific attack that merits further investigation is a malicious FPGA core that configures itself to act like a radio transmitter. It may also be possible to build a crude antenna in reconfigurable hardware, and the receiver could be another reconfigurable core, another chip on the board, or another device in the same room. In addition to reconfigurable radios, small radios, both transmitters and receivers, could also be added to the chip itself in silicon by malicious insiders in the foundry. A malicious reconfigurable core could then connect to this hard-wired radio. Researchers have already considered the problem of the temperature of the FPGA as a covert channel medium [1].

8.9 Design Theft and Failure Analysis

Although industry has invested heavily in the research and development of bitstream encryption and authentication, these mechanisms are not yet strong enough to resist attacks by determined, well-funded, state-sponsored adversaries. Infiltration or bribing of employees of FPGA manufacturers can easily bypass the current protection technology. Bitstream encryption and authentication mechanisms require many of the same kinds of secure engineering approaches (security architectures, key management, vulnerability analysis, formal verification, tamper resistance, side channel mitigation, etc.) as the designs that they are supposed to protect. Although PUFs are a promising way for generating unique keys, more work is needed to generate PUFs with sufficient entropy in a reliable way, and to generate them on FPGAs.

In order to build highly trustworthy systems on FPGAs, it will be necessary to conduct a comprehensive analysis of the failure modes (if they exist) of the specific FPGA(s) to be used in the design. For example, in response to a given fault, a system may simply halt, fail secure, continue to run, operate in a reduced functionality or maintenance mode, or recover. Therefore, an analysis of all of the states and transitions is needed. Such an analysis could show, for example, whether it is possible to transition from maintenance mode to system halt. Furthermore, it is necessary to understand how the larger system is informed of the failure of an individual component. The selection of the specific model of FPGA to be used in the design is a very important design consideration because security features such as IP protection mechanisms differ across vendors (Xilinx, Altera, Actel), FPGA types (antifuse, flash, SRAM), and specific models of FPGA (Virtex 4, Virtex 5, Virtex 6, Stratix II, Stratix III, etc.).

8.10 Partial Reconfiguration and Dynamic Security

Further research is needed to apply the results of dynamic security research to FPGA systems that use partial reconfiguration. For example, whether a system allows the policy to change, which subject changes the policy, the frequency of policy changes, whether the policies are predetermined or generated at runtime, whether it is possible to return to an earlier policy, whether the system always transitions to a monotonically more restrictive policy, whether security mechanisms can be reconfigured in addition to cores, and whether the system is hot-swappable should be considered with respect to their impact on the security and performance of partial reconfiguration. Research is also needed to allow partial reconfiguration to work without disabling encryption and to protect the ICAP interface.

8.11 Concluding Remarks

The widespread use of reconfigurable hardware in critical systems forces the FPGA community to adopt design practices that give a high priority to security. Implicitly trusting the hardware ignores the fact that hardware can behave maliciously. Designing a reconfigurable hardware based embedded system resembles software design in many ways, including the use of complex tool flows and the reuse of source code. Flaws in design tools and IP cores can be exploited to attack systems.

A wide variety of attacks against FPGAs are possible. The goals of these attacks include theft of intellectual property, theft of confidential data and keys, unauthorized modification of the hardware, and denial of service. At the foundry, malicious employees can introduce extra functionality or steal the design. Luckily, FPGAs mitigate this problem, since the design is loaded onto the chip after fabrication in a secure facility. To prevent the theft of the design after the programming of the FPGA, manufacturers have incorporated bitstream decryption mechanisms so that the design can be stored in encrypted form when the device is powered off. Despite significant effort to develop bitstream protection mechanisms, determined attackers will still try to circumvent them by performing attacks on the decryption mechanisms themselves in order to obtain the key. Preventing a determined adversary from performing a physical attack on a device is extremely difficult and costly, as decades of experience attempting to build tamper-proof hardware has shown. In addition to physical attacks, adversaries can attempt to circumvent authentication mechanisms that are supposed to prevent unauthorized parties from updating an FPGA's configuration remotely.

To minimize the damage that a flawed software program can inflict, general-purpose systems employ protection mechanisms to logically separate software processes, including rings, process address space, memory protection, virtual memory, and separation kernels. To minimize the damage that a flawed hardware core can inflict, this book has described a strategy of physical isolation of cores using moats, together with drawbridges as a means of controlled sharing between cores. Moats are implemented by constraining the placement of cores during the layout phase of design. Drawbridges are implemented by statically tracing the reconfigurable design. An on-chip reference monitor also provides separation of cores with respect to memory by enforcing a policy expressed in a specialized language. The reference monitor design flow resembles Electronic System Level (ESL) Design in that a high-level policy specification is automatically translated to a low-level hardware implementation. Policy enforcement mechanisms must be scalable in the complexity of the policy they enforce and must also be efficient in terms of area, cycle time, and overall system performance. Therefore, these methods were evaluated on a multi-core embedded design consisting of two domains that share an AES encryption core.

Proper security analysis must take a holistic view of how the FPGA fits within the larger system. This book has introduced several abstractions for managing complexity to facilitate security analysis. The reference monitor abstraction separates functional components from security components, which must be small, tamperproof, and non-bypassable. A security architecture specifies the organization of security and functional components for enforcing a specific policy. The policy abstraction is a means of specifying the subjects and objects of the system as well as the conditions under which certain subjects may access certain objects. Multiple complementary security techniques should be used, since each technique has its unique advantages and limitations, although multiple weak security mechanisms combined together do not necessarily form a stronger security mechanism. Secure systems engineering requires vigilance throughout the entire lifecycle of the system, including requirements analysis, design, implementation, testing, delivery, configuration, operation, maintenance, and audit. Security methods must be usable to both system designers and end users.

Future research is needed to improve the security of design flows and IP cores, including the development of small, trusted tools and configuration management of tools and cores. Research is also needed to analyze the security properties of hardware modules in a disciplined manner, to apply anomaly detection to hardware behavior, to formally analyze HDL specifications of IP cores, to detect and mitigate covert channels in FPGA designs, and to perform rigorous formal security analysis of bitstream protection mechanisms.

Many of the design techniques for improving security for FPGA systems can also improve the security of ASIC systems. In fact, FPGAs provide an ideal evaluation platform for prototyping novel hardware security techniques. Since hardware manufacturers are often reluctant to incorporate security methods developed by researchers, the FPGA provides a proving ground for these enhancements. FPGAs can now hold as many as eight full PowerPC soft processor cores, making them ideal for conducting experiments on security enhancements for chip multi-processors.

Two fundamental questions for any hardware-oriented security design are: what policy should be enforced on the chip, and how to enforce that policy. Cores can be arranged into equivalence classes, and a security architecture can specify which cores belong to which equivalence classes. Policy enforcement mechanisms ensure that cores belonging to different domains cannot interfere with each other. Processors with large numbers of cores are already on the drawing board, and managing communication in these designs in an efficient and secure manner is essential. Communication resources will dominate the area of the chip, and adapting the drawbridge technique to future interconnect designs will enable the controlled sharing of large numbers of cores.

References

1. J. Brouchier, T. Kean, C. Marsh, D. Naccache, Temperature attacks. IEEE Secur. Priv. **7**(2), 79–82 (2009)
2. M. Gordon, Validating the PSL/Sugar specification language using automated reasoning. Form. Asp. Comput. **15**(4), 406–421 (2003)
3. J. Hannan, F. Pfenning, Compiler verification in LF, in *Proceedings of the 7th Annual IEEE Symposium on Logic in Computer Science (LICS)*, Santa Cruz, CA, June 1992
4. W.A. Hunt, Microprocessor design verification. J. Autom. Reason. **5**(4), 429–460 (1989)
5. C.E. Irvine, K. Levitt, Trusted hardware: can it be trustworthy?, in *Proceedings of the 44th Annual Design Automation Conference (DAC)*, San Diego, CA, June 2007
6. T.E. Levin, C.E. Irvine, T.V. Benzel, G. Bhaskara, P.C. Clark, T.D. Nguyen, Design principles and guidelines for security. NPS Technical Report NPS-CS-07-014, Naval Postgraduate School, Monterey, CA, 21 November 2007
7. H.G. Rice, Classes of recursively enumerable sets and their decision problems. Trans. Am. Math. Soc. **74**, 358–366 (1953)
8. H. Sasaki, A formal semantics for Verilog-VHDL simulation interoperability by abstract state machine, in *Proceedings of the Conference on Design, Automation, and Test in Europe (DATE)*, Munich, Germany, March 1999
9. D. Spinellis, Reliable identification of bounded-length viruses is NP-complete. IEEE Trans. Inf. Theory **49**(1), 280–284 (2003)
10. K. Thompson, Reflections on trusting trust. Commun. ACM **27**(8), 761–763 (1984)

Appendix A
Computer Architecture Fundamentals

Abstract No book dealing with FPGAs and embedded systems would be complete without a discussion of computer architecture. To better understand the hardware-oriented security methods presented in this book, this appendix discusses the fundamental concepts of computer architecture that are applicable to FPGAs, ASICs, and CPUs.

A.1 What Do Computer Architects Do All Day?

Computer architects map applications to physical devices, and the requirements of the application dictate the design of the architecture. For some applications, a general-purpose CPU architecture is adequate, and the application can be implemented in software. However, sometimes a generic CPU is not enough, and custom hardware is required to deliver the necessary performance or other requirements. Examples of applications requiring custom hardware include networking and graphics, which have high throughput requirements. In addition, embedded systems often require custom hardware due to their resource-constrained nature. It is the computer architect's job to determine the optimal architecture for a particular application by balancing tradeoffs. Among the many tools in the computer architect's arsenal are measurement and metrics, cost-benefit analysis, simulation, standard programs called *benchmarks*, and logical analysis. Computer architects perform experiments that vary the parameters of the design. Since the list of possible experiments to try is always much larger than the available time, the computer architect must determine the most important parameters of the design and strip away all others.

Although designing custom hardware is hard work, large performance gains over general-purpose systems can be realized. Custom designs can be developed for many disciplines, including machine learning and neuroscience, biometrics, medicine [17, 18], cryptography [29], security, networks, computer vision, and program analysis. For a particular application, the computer architect performs analysis to determine the most common operations and then optimizes them. Understanding the structure of the problem makes it possible to extract parallelism from the application by performing common operations on multiple hardware units in parallel.

T. Huffmire et al., *Handbook of FPGA Design Security*,
DOI 10.1007/978-90-481-9157-4, © Springer Science+Business Media B.V. 2010

It is an exciting time to be a computer architect. In the words of Mark Oskin, "Seven years ago, when I started as a young assistant professor, my computer science colleagues felt computer architecture was a solved problem. Words like 'incremental' and 'narrow' were often used to describe research under way in the field.... From the perspective of the rest of computer science, architecture was a solved problem" [19] (p. 70). However, there is renewed hope for the future of the field. Techniques for improving performance that have worked in the past have run into technical "brick walls," including the "Power Wall," the "Memory Wall," and the "ILP Wall" [1]. In addition, design complexity and reliability for large, out-of-order processors presents challenges for implementation and verification. Industry has embraced chip multi-processors (CMPs) to address these problems, but developing multithreaded software that can achieve performance gains by making use of the multicore hardware has been a hard problem for decades. As researchers from the Berkeley Par Lab explain, "Industry needs help from the research community to succeed in its recent dramatic shift to parallel computing" [2] (p. 56). Rather than trying to parallelize existing scalar software code, one sensible strategy is to think about the problem from the standpoint of *what can I do with all of these cores* that cannot be done with unicore mechanisms? Making it easy for programmers to exploit multicore hardware is an open research challenge [7]. In addition to the programming problem, realistic multicore benchmarks for evaluation are also needed. Oskin concludes, "In my lifetime, this is the most exciting time for computer architecture research; indeed, people far older and wiser than me contend this is the most exciting time for architecture since the invention of the computer" [19] (p. 78).

A.2 Tradeoffs Between CPUs, FPGAs, and ASICs

Figure A.1 shows the relative generality of CPUs, FPGAs, and ASICs. There are several tradeoffs between CPUs, FPGAs, and ASICs, including software vs. hardware, generality vs. performance, cost vs. performance, and generality vs. security. CPUs run software, which is relatively inexpensive to program but comes with high overhead. ASICs are custom hardware, they have relatively high performance, and they are relatively expensive. FPGAs are more difficult to program than CPUs, but they are less expensive than ASICs. In addition, FPGAs can achieve higher throughput than CPUs but not as high as ASICs. The gap between FPGAs and ASICs is narrowing because FPGAs can be built more economically in the latest feature size, while lower-volume ASICs are sometimes fabricated with larger feature sizes, which are cheaper.

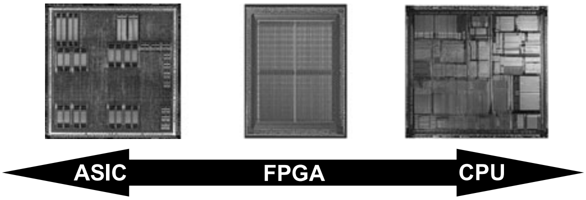


Fig. A.1 On a continuum of generality, CPUs are the most programmable, and ASICs are the least

The tradeoff between generality and security is more complex than the other tradeoffs. Attackers like generality because they can run malware on general-purpose systems. However, this doesn't mean that ASICs are free of security problems. If that were the case, *application-specific hardware* would solve the world's security problems. If a device is hard-wired to do only one thing, how can it be hacked? Sadly, this type of thinking is an example of *security through obscurity.* Application-specific devices are useful to the security community, and limiting the functionality of a system can benefit security. However, a system's security should not rest entirely upon the fact that a system has been designed to perform a limited number of functions or that the design is kept secret.

Another aspect of the tradeoff between generality and security involves the trusted foundry problem. An ASIC's intellectual property is vulnerable to theft and malicious inclusions because the design is typically sent to a third-party foundry. CPUs and FPGAs, on the other hand, are programmed after fabrication. However, one problem is replaced by another because of the risk that the hardware design may be stolen from a fielded FPGA (by circumventing the bitstream decryption mechanisms) or that the software may be stolen from a fielded general-purpose system. Stealing the design from a fielded ASIC is a much more expensive task, requiring reverse-engineering and physical sand-and-scan attacks.

A.3 Computer Architecture and Computer Science

Computer architecture spans all of computer science, and it is just as essential as theory, algorithms, languages, operating systems, networking, and security. Many of the performance gains in processors are the result of progress in algorithm development, not just the ability to fit more and more transistors on a single die. Transistors are only able to compute when they are arranged in a large-scale, correct, efficient, and fault-tolerant computer architecture. With the number of transistors on a single chip approaching and even exceeding one billion, the orchestration of such a large number of individual elements requires complex algorithms for optimization, scheduling, placement, routing, verification, branch prediction, cache replacement, instruction prefetching, control, pipelining, parallelism, concurrency, multithreading, hyperthreading, multiprocessing, speculation, simulation, profiling, coherence, and out-of-order execution, to name a few. Understanding how processors are designed and what is inside a modern processor are fundamental questions of computer architecture.

Processors are the most complicated things that humans build. Complexity is the reason that a majority of engineers at chip manufacturing companies are verification engineers, with the design engineers making up a minority. With the increasing non-recurring engineering (NRE) cost of chip manufacturing for smaller and smaller feature sizes, a mistake in the design of a chip can be fatal for even a large company. Since the number of possible states for a billion transistor design is astronomical, verification requires advanced algorithms, and its difficulty is proportional to design complexity.

A.4 Program Analysis

Computer architects study the question of what makes computers go fast. To write fast programs, it is necessary to profile what is going on inside the processor. In a modern processor, a huge number of events occur every second, on the order of one billion, and analyzing these events in real time is no small task.

A.4.1 The Science of Processor Simulation

Simulation is an essential program analysis technique, and processor simulation encompasses an entire field of study. Computer architects consume a lot of time and processor cycles running simulations. Not only is simulation useful for analyzing the performance of a processor as it runs a program, but simulating the processor is also necessary to verify its correctness. Just as an architect must create blueprints before breaking ground, a processor must be simulated prior to fabricating a chip. In between simulation and fabrication, the additional step of prototyping and testing the design on an FPGA is often taken. Simulation of the design should always be performed prior to FPGA implementation (due to the large effort required to prototype a design on an FPGA) and chip fabrication (due to the high NRE cost of fabrication).

The choice of simulation technique depends on the problem that needs to be solved. Simulation at a fine level of granularity is computationally intensive. For example, cycle-level simulation, in which events at the granularity of a single clock tick are duplicated by a software program running in a different hardware environment, is very detailed and therefore computationally intensive. Instruction-level simulation (a.k.a. *emulation*), in which events at the granularity of an instruction are represented in another processing environmet, is less expensive than cycle-level simulation but is less accurate. Emulation is useful for narrowing in on a good idea because it is fast and simple but less useful for deeper analysis because it is less accurate. For example, emulation is useful for determining the number of memory accesses and cache misses for a particular benchmark program. All simulators, both cycle-level and instruction-level, run at a much lower speed than the processors they simulate.

A key simulation parameter is how long and what to simulate. Since it is usually infeasible to simulate every event of a long-running program, it is necessary to choose a sample that represents the most pertinent behavior of the program. Taking this sample from the very beginning of the program may be a bad idea because computer programs usually begin by zeroing out arrays. Often, the critical action occurs towards the middle of the program. A sample size on the order of one hundred million instructions is common with instruction-level simulation. For cycle-level simulation, a smaller sample size is needed. Another key simulation parameter is the baseline or benchmark used for the experiment. Benchmarks are standard programs for comparing processors. The program is run by the processor being simulated.

It is extremely important to document the simulation parameters when describing the methodology of the experiment in order to understand the meaning of the results.

To study the effects of changing various aspects of the processor being simulated, computer architects modify the source code of the simulator so that the new behavior can be observed. One way of accomplishing this observation is to generate trace files and then analyze them later offline, an approach called *trace-driven* simulation. In trace-driven simulation, the simulator is modified to write specific events to a *trace file*. Trace files can get very large, on the order of gigabytes for one second of program execution, and writing to trace files slows down the simulation further. After the trace file is generated, the computer architect uses a different program to analyze the trace file off-line. For example, this program could determine the optimal cache replacement policy for a simulation that exercised different memory management strategies. Alternatively, it could determine the optimal branch prediction technique for a simulation of branch events. However, trace-driven simulation is limited by the fact that it is static, which is not useful for studying behaviors that involve significant speculative execution [30].

SimpleScalar is a suite of uni-processor simulators and tools for compilation [4]. The suite consists of *SIM–FAST*, a functional simulator that provides no timing, *SIM–OUTORDER*, a functional and timing simulator with a detailed timing model, *SIM–CACHESIM*, for simulating memory behavior, and *SIM–BPRED*, for simulating branch prediction. SimpleScalar can run programs with different ISAs, where the *MACHINE.DEF* file specifies the ISA, which can be Pisa, Alpha, Arm, or x86. Pisa is a made-up architecture based on MIPS, and a specific version of gcc will produce Pisa code. A real compiler and a set of binaries is available for Alpha. Arm is primarily used in embedded systems, and x86 is ubiquitous. *SIM–MAIN* is the main simulator loop.

A.4.2 On-Chip Profiling Engines

Because simulation typically slows down execution by at least one order of magnitude, capturing and analyzing events in real-time using on-chip profiling hardware is very useful for computer architecture research. An on-chip profiling module can capture and analyze events in real time and at high bandwidth without the need for dumping every event to a trace file on the hard disk. Although some processors have profiling features such as special profiling registers, more full-featured profiling engines desired by computer architecture researchers are rare because of economic priorities. Chip manufacturers have their hands full verifying the processor design itself without the extra task of designing and verifying a profiling module, and most end users will never use the profiling module, not being computer architecture researchers.

To overcome this problem, one option is to employ a co-processor for off-loading computationally-intensive analysis tasks. The co-processor can be a chip on the

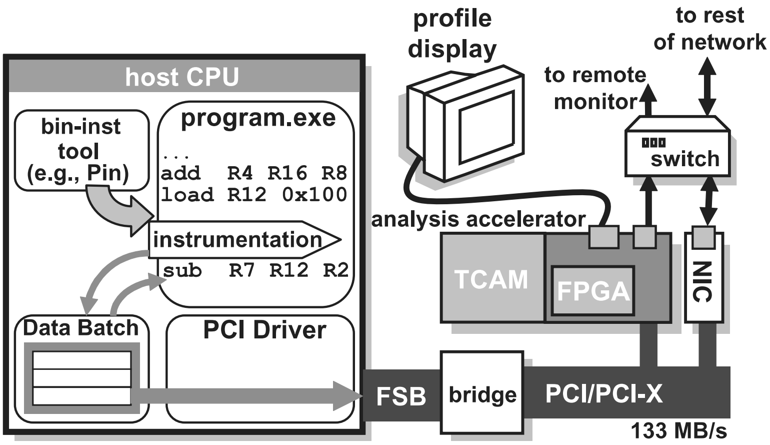


Fig. A.2 Online program analysis architecture. Computationally-intensive program analysis tasks can be off-loaded to an FPGA co-processor

same board as the main processor, or it can be an FPGA board that is connected to the Peripheral Component Interconnect (PCI) interface of the motherboard, as shown in Fig. A.2. As the processor executes the program, software instrumentation gathers profile data, which is batched in a buffer. Rather than performing computationally-intensive analysis of the profile data in software on the host CPU, the data is then written to the PCI driver. Next, the FPGA analysis module processes the profile data, and the output is used by an optimizer, a human operator, a display, or a remote monitoring unit. Clearly, this online program analysis architecture has lower bandwidth than an on-chip profiling module. An approach that can achieve higher bandwidth applies 3-D integration, an established technology in which a commodity integrated circuit is enhanced with a separate chip after fabrication [15, 16]. This additional integrated circuit contains a profiling module for analyzing events on the commodity chip. The application of 3-D Integration to security has also been proposed [14]. In this approach, specialized security functions reside in one IC, called the *3-D control plane*, which monitors and enforces a security policy on the commodity IC, called the *computation plane*.

A.4.3 Binary Instrumentation

Binary instrumentation is another useful program analysis technique [12] and is used by itself or, for example, as a component of on-chip profiling engines, as shown in Fig. A.2. An unmodified binary can be instrumented with calls to custom functions in response to specific events. These software functions can perform analysis of the event or simply write specific details of the event to a trace file. Like sim-

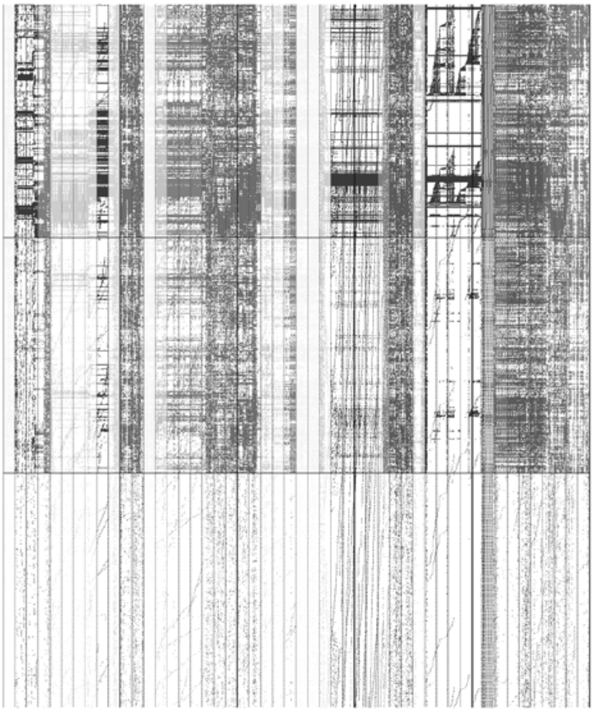


Fig. A.3 A plot of the cache behavior of Firefox over fifty million instructions as it loads a web page. The x-axis is time, and the y-axis is a function of the address of the memory access. Each *vertical slice*, or interval, represents one million instructions. The *top band* shows L1 cache hits, the *middle band* shows L1 cache misses, and the *bottom band* shows L2 misses. Intervals are colored according to the wavelet-based phase classification algorithm [13]

ulation, there is at least one order of magnitude slowdown, and writing to trace files slows things down even further. However, binary instrumentation provides the opportunity to study the complex, multi-threaded behavior of applications like web browsers, word processors, and graphics editing programs. Specific features of these applications can be invoked by interacting with the user interface (e.g., invoking menu commands, dialog boxes, etc.)

A.4.4 Phase Classification

Phase analysis is another useful tool in the computer architect's arsenal [21, 22]. Computer programs exhibit repeating behaviors over the course of their execution. Identifying these phases, which are time intervals that share similar behavior, provides several opportunities for guiding run-time optimizations and reducing simulation time [8, 9, 20, 23]. Phase classification works by counting the frequency each basic block[1] is executed during each time interval. A Basic Block Vector (BBV) is simply an array with one entry for each basic block that stores the frequency that each basic block is executed, weighted by the number of instructions in the basic block and normalized by the total number of basic blocks executed during the interval. A similarity metric called *Manhattan Distance* is used to compare BBVs, and it is the sum of the absolute value of the difference between each element of two BBVs. Random Projection is used to reduce the dimensionality of the data, and k-means clustering is used to group the BBVs into clusters. All BBVs in a cluster belong to the same phase. Other structures for phase classification include those that capture memory access stride, structures that employ the notion of working sets, and even structures that use wavelet coefficients [13]. Figure A.3 shows the memory behavior of Firefox over fifty million instructions as it loads a web page, where intervals are colored according to the wavelet-based phase classification algorithm.

A.5 Novel Computer Architectures

Making effective use of billions of transistors and a multitude of cores on a single chip is the goal of next-generation processor design proposals. The key to success is managing complexity by using robust design abstractions that do not hide the technical nuances. Computation will become less expensive, but communication will become more expensive. Interconnection networks that manage the communication among large numbers of cores will likely consume a large portion of the on-chip resources. Just as processors use a hierarchy of memories, it is likely that future processors will use a hierarchy of interconnect.

A.5.1 The DIVA Architecture

The DIVA architecture [3] is an attempt to manage complexity. Verification of processors that use speculative execution is very computationally intensive. The key idea is that verifying the correctness of the result of a computation is less computationally demanding than computing that result. Therefore, the runtime correctness of a complex processor that uses out-of-order execution can be verified by a smaller *checker* hardware module. This checker unit's small size makes it much easier to verify. The checker unit resides on the chip along with the more complex processor that uses speculation. There are several parallels between the DIVA concept and the reference monitor concept. Both are small, making them easier to verify, both help to manage complexity, and both are run-time mechanisms that reside on the chip.

[1] A basic block is a straight-line sequence of code with one entry point, one exit point, and no jump instructions.

A.5.2 The Raw Microprocessor

To manage complexity, future processors will likely employ large numbers of computational cores. These cores will need to communicate in an efficient manner, requiring alternatives to traditional bus interconnect. A shift from computation-centric to communication-centric design is underway. The idea of networks-on-a-chip (NoC) has been around for a while [6], but few NoCs have been realized due to their complexity. Implementation of a network-on-chip requires building scaled-down networking routers that are located throughout the chip. The chip is divided into tiles, where each tile has a computational core, a network interface (NI), and a switch. Various topologies have been studied, such as the hierarchical ring-based interconnect [5], although realistic evaluation benchmarks, network traffic models, and simulation environments are currently lacking. While the primary application of NoCs is embedded systems, Intel's technology roadmap for future interconnect calls for a scaled-down network protocol stack for on-chip interconnection networks [11]. The Raw Microprocessor Architecture, developed at MIT, uses both static and dynamic routing [25, 28]. Tilera is a company that has developed a 64-core processor aimed at the embedded market. Tilera's processor is called TILE64 and is based on the MIT Raw Architecture.

A.5.3 The WaveScalar Architecture

WaveScalar [24] is a dataflow architecture that is an alternative to the von Neumann architecture. The problem with the von Neumann architecture is that all of the data, including data residing in low levels of the memory hierarchy (e.g., off-chip DRAM), has to be brought into a central location (the CPU) to be processed, before being sent back out to memory. The key idea of WaveScalar is to execute the program in place in the memory system. Whenever a variable's value changes, this automatically triggers the recalculation of the values of other variables that are dependent on it. WaveScalar exploits the *principle of locality*, meaning that if the program accesses a memory value at location i, it is likely that the program will access one of i's neighbors in the near future. The basic building block of the WaveScalar processor is the WaveCache, consisting of both memory and processing elements.

A.5.4 Architectures for Medicine

There are many medical applications of computer architecture. For example, subthreshold voltage processors in medical devices can be used to prevent blindness [17, 18]. Subthreshold voltage processors trade processor performance for energy savings. The device uses a subthreshold voltage processor to measure intraocular pressure to delay the onset of blindness in glaucoma and diabetes patients.

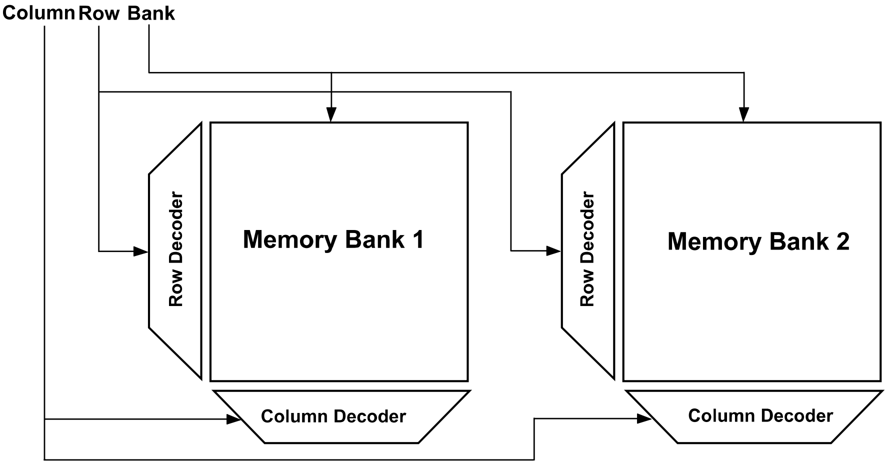


Fig. A.4 Memory is arranged into banks, with row decoders and column decoders

"The system is designed to grip the inner surface (vitreous) of the eyeball. The system will be installed via out-patient surgery, and it will provide patients with real-time feedback on the interior eye pressure. Recent medical studies have shown that careful monitoring and subsequent control of intra-ocular pressure can delay the onset of blindness in glaucoma and diabetes patients. The intra-ocular pressure measurement system includes a subthreshold sensor processor, 384 bytes of memory, 1024 bytes of ROM, a MEMOS-based pressure sensor, a Peltier-based energy scavenging mechanism which utilizes temperature gradients within the eyeball to produce electricity, and a communication system based on inductive coupling" [18].

A group at Stanford is working on a retinal prosthesis implant that is placed behind the retina and has an array of tiny solar cells, and other groups send power to their devices via RF signals [10].

A.6 Memory

A good analogy to the memory hierarchy of a computer is a kitchen. When the cook needs an ingredient, the first place to look is the refrigerator. The refrigerator is like a cache: finding the needed ingredient is analogous to a cache hit, and not finding it is analogous to a cache miss. If the ingredient is not in the refrigerator, it is necessary to check the pantry. The pantry is analogous to an L2 cache. If the required ingredient is not in the pantry, this is analogous to an L2 cache miss, and a trip to the supermarket is required. The supermarket is analogous to off-chip memory because of the relatively large number of clock cycles required to perform an access to off-chip memory. Fetching an item from the pantry takes progressively longer than fetching an item from the refrigerator in the kitchen, and driving to the supermarket takes progressively longer than fetching the ingredient from the pantry.

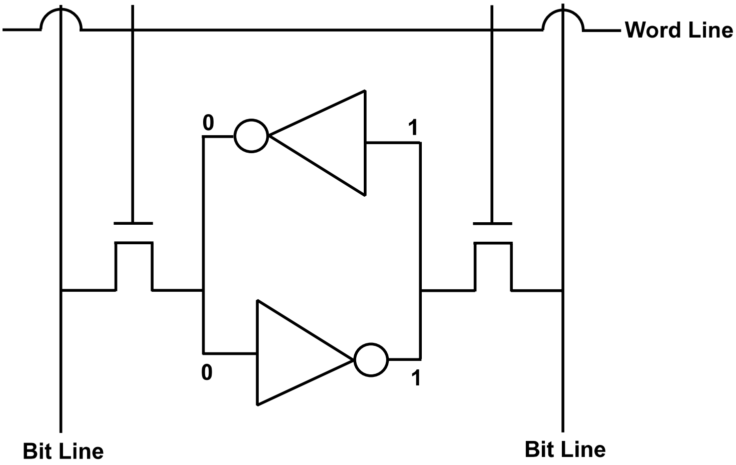


Fig. A.5 An SRAM cell stores a single bit. Feedback between two NOT gates, which enter a stable equilibrium, is the storage mechanism

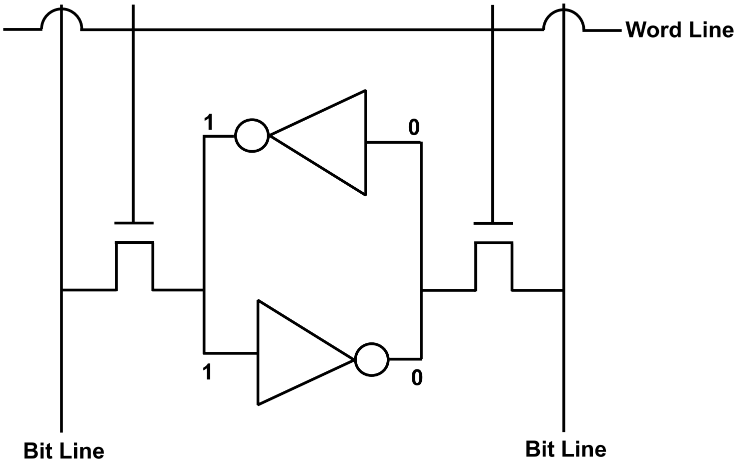


Fig. A.6 Another stable equilibrium of an SRAM cell

Memory is arranged into banks, as shown in Fig. A.4. A memory address must specify which bank along with the row and column of the desired location within that bank. A row decoder and a column decoder select the row and column specified in the address. The bit is stored at the junction of the row and column. Figure A.5 shows how a bit is stored in an SRAM cell. Feedback between two NOT gates, which enter a stable equilibrium, is the storage mechanism. Figure A.6 shows another stable equilibrium. Six transistors are needed for each SRAM cell: two for each NOT gate, and two additional transistors. Figure A.7 shows a multi-port SRAM

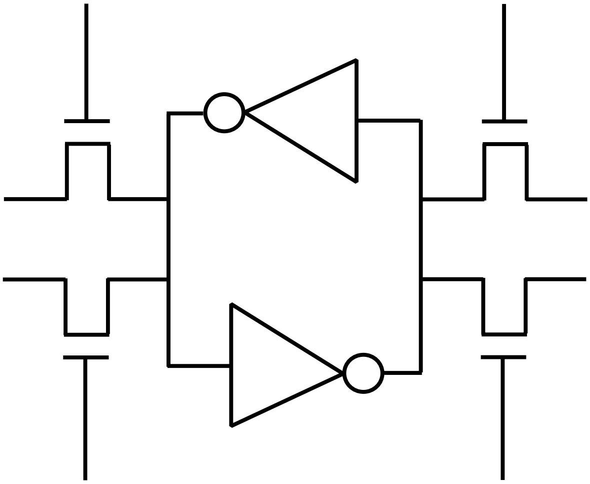


Fig. A.7 A Multi-port SRAM cell allowing multiple simultaneous access

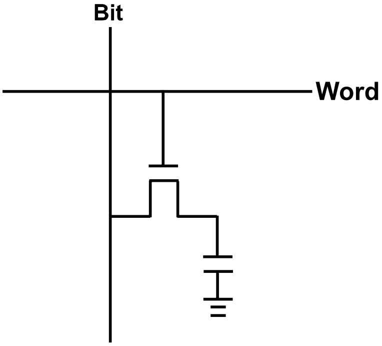


Fig. A.8 A DRAM cell

cell that allows multiple simultaneous access, which is useful for video cards that read and write at the same time or perform multiple accesses to the same location.

Despite its large power and area requirements, SRAM is used for on-chip L1 caches and registers, where high performance is needed. Off-chip DRAM, on the other hand, uses fewer transistors per bit, reducing the power and area for each bit. This makes it possible to build memories very densely but also reduces the performance (1 ns for SRAM and 100 ns for DRAM). A DRAM cell uses a capacitor, which is a *bucket* of electrons. A write to DRAM involves either charging or discharging, but a read is more tricky. Since the capacitor leaks slowly, buckets that have the value one need to be refilled. The currents involved are tiny, with just one hundred electrons representing a single bit. Figure A.8 shows a DRAM cell, where the capacitor is drawn in the lower right portion of the figure.

Due to the growing gap between processor and memory performance, system designers employ a hierarchy of memories, a concept devised by von Neumann in 1946. To capture spatial locality, caches are broken up into blocks. In a *fully associative* cache, a block can go anywhere in the cache. This requires a system of tags in order to know which pieces of memory reside in the cache, and parallel searches of all slots are required. At the other extreme is a *direct mapped* cache, which works like a hash table because a block can only go to one location. Direct mapped caches are very fast but may have collisions of addresses to the same slot. In between the two extremes is a *set associative* cache, which allows a block to go to a set of possible locations. Cache replacement policies include least recently used (LRU), least frequently used (LFU), random, and *oracle*, which makes a prediction about which pieces of memory will be used in the future.

A common program optimization technique is to change the way that a program accesses memory in order to reduce the number of cache misses. The program is redesigned so that needed data fits into the cache better and can be prefetched more effectively.

A.7 Superscalar Processors

Analysis of the dependencies of code determine when more resources (e.g., adders) are needed in order to execute multiple instructions in parallel. Multiple issue processors include *Very Long Instruction Word (VLIW)* machines and superscalar machines. In a Very Long Instruction Word (VLIW) machine, two instructions can be glued together to form one big instruction, and the compiler does the scheduling. In a superscalar machine, the processor hardware schedules the instructions. More things are being done by the hardware these days rather than the compiler. For example, *out-of-order execution*, in which the hardware dynamically rearranges instructions, is becoming common.

Tomasulo's Algorithm is a distributed, scalable algorithm for finding parallelism [26]. For data dependencies, out-of-order execution adapts dynamically to the data flow graph. In a sense, the data flow graph is sucked through a straw, and since there are finite resources on the chip, sometimes there is a miss. Tomasulo's Algorithm uses dynamic loop unrolling, mix & match, and reservation stations. Instructions wait at the reservation stations until they have what they need. After executing, the fact that they now have what they need is broadcast over the broadcast bus. Tomasulo's algorithm was implemented on the IBM 360/91, which was a commercial flop. It had four floating point registers and little compiler support. In the case of an add, the instruction waits for the two operands it needs, checking the register file. The result is broadcast everywhere over a common data bus. The reservation station scheme requires entries for the operation (e.g., addition, subtraction), the operands, the names of the reservation station of origin, and whether the reservation station is occupied.

Problems with Tomasulo's Algorithm include broadcasting to everybody over the common data bus. Since order matters, arbitration is needed to prioritize the communication. Since overloading the bus is a problem, multiple busses may be necessary or a *cross bar* network of point-to-point interconnection. Complexity is another problem with Tomasoulo's Algorithm. Tomasulo's Algorithm teaches us that good solutions are a *compromise*. An alternative scheduling scheme to Tomasulo's Algorithm is *just in time scheduling* used by the Pentium 4.

For name dependencies, the hardware implementing Tomasulo's Algorithm uses register renaming. While the user and the compiler can only see a small number of architectural registers (e.g., 32), the set of physical registers can be larger (e.g., 256), and register renaming maps the smaller set onto the larger set. This register renaming contributes to the complexity of the design and verification of Tomasulo's Algorithm.

A.8 Multithreading

Figure A.9 shows two threads executing on a superscalar processor and illustrates the problem of *vertical waste*, in which redundant hardware resources often go unused [27]. Tullsen et al. define empty issue slots as vertical waste when the processor issues no instructions in a cycle; horizontal waste occurs when all of the available issue slots cannot be filled in a cycle [27]. At one point in Fig. A.9, Thread 2 is not able to use any of the hardware resources. To mitigate this problem, one might try to interleave both threads, a scheme called *fine-grained multithreading*, but this does

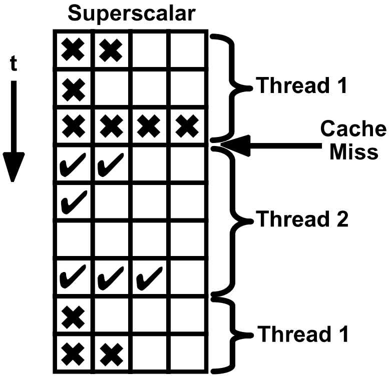


Fig. A.9 Superscalar machines exhibit *vertical waste*

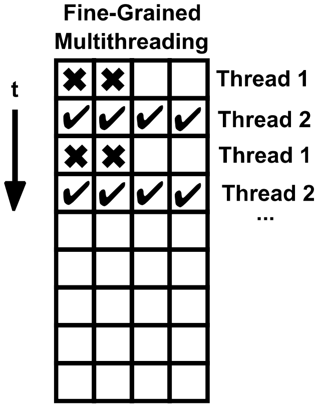


Fig. A.10 Fine-grained multithreading machines exhibit *horizontal waste*

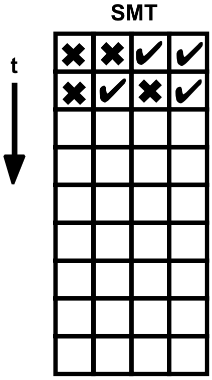


Fig. A.11 A Simultaneous Multithreading (SMT) Machine interleaves instructions every cycle, performing them simultaneously. This approach does a good job of packing the instructions in and more fully utilizing redundant hardware resources

not eliminate the *horizontal waste* of resources, as shown in Fig. A.10. Although interleaving has solved the vertical waste problem, there are still times when all of the hardware resources are not being utilized fully. Simultaneous Multithreading (SMT) machines interleave instructions every cycle, performing them at the same time, as shown in Fig. A.11. SMT machines do a good job of packing the instructions in. Technical issues of SMT machines include:

- page tables and TLBs (a TLB is a cache for the page table)
- register assignments (separate registers are required)
- multiple stacks
- the memory hierarchy (one thread might overwrite the cache)
- separate branch predictors
- parallel instruction issue logic
- sharing one structure between threads

References

1. K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, K.A. Yelick, The landscape of parallel computing research: a view from Berkeley. Technical Report No. UCB/EECS-2006-183, University of California, Berkeley, 18 December 2006
2. K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, K. Yelick, A view of the parallel computing landscape. Commun. ACM **52**(10), 56–67 (2009)
3. T.M. Austin, DIVA: a reliable substrate for deep submicron microarchitecture design, in *Proceedings of the 32nd International Symposium on Microarchitecture (MICRO-32)*, Haifa, Israel, November 1999
4. T. Austin, E. Larson, D. Ernst, SimpleScalar: an infrastructure for computer system modeling. IEEE Comput. **35**(2), 59–67 (2002)
5. S. Bourduas, Modeling, evaluation, and implementation of ring-based interconnects for network-on-chip. Ph.D. dissertation, McGill University, Dept. of Electrical and Computer Engineering, Montreal, Canada, May 2008
6. W.J. Dally, B. Towles, Route packets, not wires: on-chip interconnection networks, in *Proceedings of the 37th Design Automation Conference (DAC)*, Las Vegas, NV, June 2001

7. J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
8. L. Eeckhout, R.H. Bell Jr., B. Stougie, K. De Bosschere, L.K. John, Control flow modeling in statistical simulation for accurate and efficient processor design studies, in *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, Munich, Germany, June 2004
9. L. Eeckhout, J. Sampson, B. Calder, Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation, in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'05)*, Austin, TX, October 2005
10. W.D. Jones, A form-fitting photovoltaic artificial retina. *IEEE Spectrum*, **46**(12), December 2009. http://spectrum.ieee.org/biomedical/bionics/a-formfitting-photovoltaic-artificial-retina
11. D. Kanter, The Common System Interface: Intel's future interconnect. *White Paper, Real World Technologies*, August 2007
12. C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, K. Hazelwood, Pin: building customized program analysis tools with dynamic instrumentation, in *Proceedings of the 2005 ACM/SIGPLAN Conference on Programming Language Design and Implementation*, Chicago, IL, June 2005
13. T. Huffmire, T. Sherwood, Wavelet-based phase classification, in *Proceedings of the Fifteenth International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Seattle, WA, September 2006
14. T. Huffmire, J. Valamehr, T. Sherwood, R. Kastner, T. Levin, T.D. Nguyen, T. Sherwood, Trustworthy system security through 3-D integrated hardware, in *Proceedings of the 2008 IEEE International Workshop on Hardware-Oriented Security and Trust (HOST)*, Anaheim, CA, June 2008
15. S. Mysore, B. Agrawal, S.-C. Lin, N. Srivastava, K. Banerjee, T. Sherwood, Introspective 3D chips, in *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, October 2006
16. S. Mysore, B. Agrawal, S.-C. Lin, N. Srivastava, K. Banerjee, T. Sherwood, 3-D integration for introspection, in *IEEE Micro: Micro's Top Picks from Computer Architecture Conferences*, January–February 2007
17. L. Nazhandali, B. Zhai, J. Olson, A. Reeves, M. Minuth, R. Helfand, S. Pant, T. Austin, D. Blaauw, Energy optimization of subthreshold-voltage sensor network processors, in *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, Madison, WI, June 2005
18. L. Nazhandali, M. Minuth, B. Zhai, J. Olson, T. Austin, D. Blaauw, A second-generation sensor network processor with application-driven memory optimizations and out-of-order execution, in *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*, San Francisco, CA, September 2005
19. M. Oskin, The revolution inside the box. Commun. ACM **51**(7), 70–78 (2008)
20. E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, B. Calder, Using SimPoint for accurate and efficient simulation, in *International Conference on Measurement and Modeling of Computer Systems*, San Diego, CA, June 2003
21. T. Sherwood, E. Perelman, G. Hamerly, B. Calder, Automatically characterizing large scale program behavior, in *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, October 2002
22. T. Sherwood, E. Perelman, G. Hamerly, S. Sair, B. Calder, Discovering and exploiting program phases, in *IEEE Micro: Micro's Top Picks from Computer Architecture Conferences*, November–December 2003
23. R. Srinivasan, J. Cook, S. Cooper, Fast, accurate microarchitecture simulation using statistical phase detection, in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'05)*, Austin, TX, March 2005
24. S. Swanson, K. Michelson, A. Schwerin, M. Oskin, WaveScalar, in *Proceedings of the 36th International Symposium on Microarchitecture (MICRO)*, San Diego, CA, December 2003

25. M.B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, A. Agarwal, The RAW microprocessor: a computational fabric for software circuits and general-purpose programs. IEEE Micro **22**(2), 25–35 (2002)
26. R.M. Tomasulo, An efficient algorithm for exploiting multiple arithmetic units. IBM J. Res. Develop. **11**(1), 25 (1967)
27. D.M. Tullsen, S.J. Eggers, H.M. Levy, Simultaneous multithreading: maximizing on-chip parallelism, in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, 1995
28. E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, A. Agarwal, Baring it all to software: RAW machines. IEEE Comput. **30**(9), 86–93 (1997)
29. L. Wu, C. Weaver, T. Austin, CryptoManiac: a fast flexible architecture for secure communication, in *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA)*, Gothenburg, Sweden, July 2001
30. C. Zilles, G. Sohi, Master/slave speculative parallelization, in *Proceedings of the 35th International Symposium on Microarchitecture (MICRO)*, Istanbul, Turkey, November 2002