AFRL-RI-RS-TR-2014-105

ABC: AGING-BASED IC CONFIGURATION

UNIVERSITY OF CALIFORNIA, LOS ANGELES

APRIL 2014

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2014-105 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/ S /
GARRETT S. ROSE
Work Unit Manager

/ S /
MARK H. LINDERMAN
Technical Advisor, Computing & Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
APRIL 2014	FINAL TECHNICAL REPORT	NOV 2011 – NOV 2013

4. TITLE AND SUBTITLE

ABC: AGING-BASED IC CONFIGURATION

5a. CONTRACT NUMBER
N/A

5b. GRANT NUMBER
FA8750-12-2-0014

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Miodrag Potkonjak

5d. PROJECT NUMBER
T2HW

5e. TASK NUMBER
UC

5f. WORK UNIT NUMBER
LA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of California, Los Angeles
3532G Boelter Hall
Los Angeles, CA 90095

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2014-105

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The continuously widening gap between the non-recurring engineering and recurring engineering costs of producing integrated circuit products in the past few decades gives high incentives to unauthorized cloning and reverse-engineering of ICs. Existing IC digital rights management schemes often demands high overhead in area, power, and performance, or require non-volatile storage. Our goal is to develop a novel intellectual property protection technique that offers universal protection to both ASICs and FPGAs from unauthorized manufacturing and reverse engineering. We present a proof-of-concept implementation of the basic elements of our techniques, as well as a case study of applying the anti-cloning technique to a nontrivial FPGA design. Furthermore, we present the use of benign hardware Trojans (BHTs) as a security measure for an embedded system with a software component and a hardware execution environment. Based on delay logic, process variation, and selective transistor aging, the BHT can be incorporated into an embedded system for the software and the hardware components to authenticate each other before functional execution. We demonstrate an implementation of such a BHT within an embedded system on a Xilinx Spartan-6 FPGA platform. Using the same platform we will also show that the BHT security measurement has a low to modest amount of performance overhead basing on the test results from a variety of synthetic and real world benchmarks.

15. SUBJECT TERMS

IP protection, active hardware metering, unclonability, hardware Trojans

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**GARRETT S. ROSE**
U	U	U	UU	48	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 SUMMARY

The continuously widening gap between the non-recurring engineering (NRE) and recurring engineering (RE) costs of producing integrated circuit (IC) products in the past few decades gives high incentives to unauthorized cloning and reverse engineering of ICs. Existing IC digital rights management (DRM) schemes often demand high overhead in area, power, and performance or require non-volatile storage. We develop a novel intellectual property (IP) protection technique that offers universal protection to both application-specific integrated circuits (ASICs) and field-programmable gate arrays (FPGAs) from unauthorized manufacturing and reverse engineering. In this project we have shown a proof-of-concept implementation of the basic elements of the technique, as well as a case study of applying the anti-cloning technique to a nontrivial FPGA design. Due to its moderate size and design sophistication, we have chosen to use the LEON3 processor as a platform to demonstrate a simple anti-cloning mechanism using SR arbiters. LEON3 is a 32-bit general-purpose processor based on the SPARC V8 architecture. The complete VHDL source code of the LEON3 processor is released under the GNU Public License (GPL) for academic use.

We have also developed an approach that uses benign hardware Trojans (BHTs) as a security mechanism for an embedded system with a software component and a hardware execution environment. Based on delay logic, process variation, and selective transistor aging, the BHT can be incorporated into any FPGA- or ASIC-based embedded system for the software and the hardware components to authenticate each other before functional execution. We have demonstrated an implementation of such a BHT within an embedded system on a Xilinx Spartan-6 FPGA platform. Using the same platform we also show that the BHT security measurement has a low to modest amount of performance overhead based on the test results from a variety of synthetic and real-world benchmarks. We have implemented the BHT on an open-source processor, specifically OpenRISC OR1200, a 5-stage 32-bit general-purpose processor. As our software environment we used the common GNU toolchain for 32-bit C/C++ support, including GCC, binutils, and GDB.

2 INTRODUCTION

There exist several high-impact gaps in the design, implementation, and manufacturing of integrated circuits, including silicon capacity vs. design productivity, number of gates vs. number of pins, and the disparity between gate delays and wire delays. These gaps have been having deep and profound impacts on both IC design and manufacturing processes. In the last two decades another gap emerged that may have a far-reaching economic impact on the semiconductor industry. The gap between non-recurring engineering costs and recurring engineering costs has been growing exponentially as the manufacturing process continues to scale down. The numbers are truly fantastic: while in the sixties the cost of manufacturing one gate was $1, it is expected that by the end of this decade 1 trillion gates will cost only $1. Meanwhile, owing mainly to the increasing size and verification complexity of the designs that are executed, the cost of designing a modern IC product has skyrocketed. On the other hand, the cost of building a state-of-the-art semiconductor foundry has also rapidly grown to well over $1 billion. The NRE-RE cost gap provides high incentives for independent silicon foundries to recuperate setup costs by manufacturing non-authorized ICs, and for fab-less design houses to prevent manufacturing piracy.

This situation provided impetus for the initiation of active IC digital rights management research. Several techniques have been proposed and implemented. They share at least one of two common denominators: the use of physically unclonable functions (PUFs) [1] [2] or conditionally enabling through classical cryptographical one-way functions [4] [5]. While these techniques address several aspects of IC intellectual property (IP) protection such as prevention of use of non-authorized ICs, they have significant limitations including rather high area, power, and frequency overheads, additional storage requirements for enabling keys, and susceptibility to operational and environmental conditions.

Most importantly, they do not offer protection of the design know-how that is often strategically important, i.e. although the attacker may not be able to produce non-authorized ICs, he can gain insight on how significant parts of the design are created.

Furthermore, the arsenal for IP protection is even more sparse for volatile SRAM-based FPGA designs [6] [7] [8], where a locally-stored configuration bitstream is usually required. Although the stored bitstreams can be encrypted with existing cryptographical mechanisms, there

exists similar overhead and storage concerns over this technique as crytptographically enabled IC designs.

In this report, we present a novel technique for comprehensive IP protection and active device metering with very low resource and energy overhead. The technique is universally applicable to both ASIC and FPGA designs. In addition to protecting against non-authorized manufacturing, we show that reverse-engineering of the design can be prevented at user-specified levels. Key to this IP protection approach is to implement sensitive logic paths in such a way that their functionality can be altered, post-silicon, using targeted device aging. In the context of FPGA IP protection, this means that the chosen sensitive logic paths can be altered, through targeted aging, independent of the bitstream design.

The notion of sensitive logic paths reflects the understanding that not the entire silicon product requires protection against cloning or reverse engineering. For instance, it is typically not necessary to protect a generic multiplier as such designs are easily obtainable or trivial. Sensitive logic paths refer to the portions of the design where critical know-how or functionality is embedded, such as the Finite State Machine (FSM) of a video compression engine.

3 METHODS, ASSUMPTIONS, AND PROCEDURES

3.1 Securing Netlist-level FPGA Designs

3.1.1 Key Concepts

To understand how logic components can be configured post-silicon or post-bitstream, we introduce the concept of delay logic. Traditional binary combinational logic produces outputs that can be determined statically once the functions and the connectivities of each element, e.g. the netlist of the design, are known. Delay logic is a type of logic whose outputs depend on a third runtime factor, which consists of the delays of the gates in the circuit.

A small example is presented in Figure 1 and Tables 1 and 2, where the outputs of an XOR gate and an OR gate are combined by an arbiter element. The output C of the arbiter is determined in the following fashion:

- If a rising edge arrives on input port 0 before input port 1, then the output is 0.
- If a rising edge arrives on input port 1 before input port 0, then the output is 1.

If the initial state of the circuit is that A and B are both 0, and A and B are changed at the same time, then it is clear that the output port C depends on the knowledge of the delay of the XOR and OR gates in the circuit. Tables 1 and 2 show the values of output C as a function of inputs A and B for the two possible relative speeds of the XOR and OR gates. Evidently the whole circuit behaves like an AND gate when the XOR is faster than the OR gate, and an OR gate when the OR is faster than the XOR.

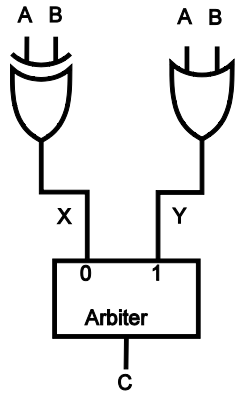

Figure 1: Delay logic example using XOR and OR gates.

Table 1: Truth table when XOR is faster than OR in Figure 1.

A	B	X	Y	C
0	0	0	0	0
0	1	1	1	0
1	0	1	1	0
1	1	0	1	1

Table 2: Truth table when OR is faster than XOR in Figure 1.

A	B	X	Y	C
0	0	0	0	0
0	1	1	1	1
1	0	1	1	1
1	1	0	1	1

Delay logic has a clear advantage to the traditional combinational logic for protection against cloning and reverse engineering, due to the fact that its output depends on a dynamic delay factor. In the context of this paper, we assume that obtaining such information from arbitrary gates and designs is difficult. However, there are challenges to using delay logic in implementation. Many factors, such as die temperature, V_{dd}, and manufacturing variations, affect the logic delay, thus making it difficult to predict. The use of the arbiter elements relaxes the

design constraints by measuring only the relative delays between two paths. Furthermore, the reliability of the delay logic can be improved through post-silicon configuration.

Negative Bias Temperature Instability, or NBTI [9] [10], is an aging process that occurs to CMOS transistors when a negative bias is applied to the gate. When stressed, the breakage of hydrogen-silicon bonds creates interface traps which lead to increases in the effective threshold voltage V_{th} of the gate. Figure 2 shows a typical CMOS representation of a NOR gate. The top transistors are of PMOS type, and the bottom ones are of NMOS type. Although NBTI acts upon both the PMOS and NMOS transistors, PMOS transistors are impacted more significantly than NMOS transistors, as they are always negatively biased when turned on. Nevertheless, the overall effect of the NBTI aging is that the logic propagation delay through the gate is increased, i.e. the gate becomes slower.

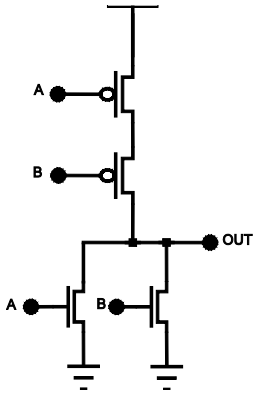

Figure 2: CMOS NOR gate.

Combining selective NBTI aging with arbiter-based delay logic yields a new approach to protecting the sensitive logic paths of an IC design. Without the knowledge of relative logic speeds, an attacker cannot understand the functionality of the circuit, even if he is equipped with the full gate-level netlist. Without proper post-silicon configuration, a non-authorized copy of the IC will not function as intended.

3.1.2 Related Efforts and State-of-the-Art

Security techniques for FPGA designs and implementation are a broad research area that covers a variety of issues ranging from digital rights management and detection of malicious circuitry to reverse engineering and trusted synthesis. There are several recent surveys on FPGA security [11] [12]. In our brief survey of the related work we mainly focus on directly related IC DRM and FPGA security techniques.

The first set of FPGA DRM techniques was created by John Lach and his coauthors [13] [14] [15]. Champagne et al. [16] discussed secure techniques for distribution of FPGA configurations.

The largest motivator for studying IC reverse engineering is its surprising easiness and effectiveness of such attacks as reported by Cambridge University researchers [17]. Consequently, several groups demonstrated that even highly secure hardware security primitives such as PUFs are surprisingly easy to reverse engineer [18] [19].

Many IC IP protection efforts emphasize techniques that enable zero knowledge proofs that a particular hardware is designed by a specific entity. Almost all of them use some form of design watermarking and/or fingerprinting [20] [21] [22]. Consequently, these IC fingerprinting techniques were combined with data mining techniques to form passive metering approaches [23] [24] [25] [26]. Passive IC metering enables counting the number of non-authorized ICs. Since 2007, several active IC metering techniques have been developed [3] [4]. These techniques enable remote activation and deactivation of ICs. In many of these techniques PUF and PPUF [27] [28] play a crucial role in the creation and employment of unique IC IDs. They are creative and effective solutions and certainly advance the active metering research frontier. Nevertheless, they are subject to several significant limitations such as high hardware and energy overheads, limited security protocols flexibility (e.g. no mechanisms for specifying active time intervals and only single user control), and the requirement of key storage that may be the source of security vulnerabilities. In addition, they are not amenable to quantitative security analysis and do not guarantee prevention of IC reverse engineering.

While recovering netlists by reverse engineering actual ICs is a well-established research and business endeavor [29] [30] [31], there is surprisingly little reported work on IC reverse engineering to higher levels of abstraction. Notable exceptions include efforts at the University of Michigan [32], Michigan State [33] [34] [35], and recently the Air Force Institute of Technology [36] [37] [38]. However, there have been numerous reverse engineering efforts at lower levels of abstraction mainly with the goal of verifying actual implementations [39] [40] [41].

Recently, in a series of papers Torrance and James provided a detailed description about the capabilities and limitations of state-of-the-art industrial IC reverse engineering procedures [42] [43]. Even more recently, reverse engineering has started to attract rapidly growing interest from the academic community [44].

In addition to directly related IC structure extraction and reverse engineering techniques, there are several other related areas. They are related either because we use their techniques and tools or due to conceptual similarity. The most difficult task of IC reverse engineering is probably FSM extraction and traversal. The problem has been addressed in several communities [45] [46] [47]. Furthermore, identification and coverage of regular patterns has been a popular and important problem in behavioral and system synthesis [48] [49].

3.1.3 The SR Arbiter

In this section, we will introduce a delay logic element with the combined functions of an arbiter and two NAND gates that compete for the control of the arbiter output. We will also show how it can be implemented on an FPGA platform with repeatable results.

D-type flip-flops (DFFs) have been used as arbiters in PUF designs in the past [27], where the path differences are designed to be offset by the preceding tuning circuits. Though easy to implement, arbiters implemented using DFFs have a built-in bias due to the fact that the Clock-to-Q and D-to-Q paths are different by design. Figure 3 shows an example of an asymmetrical DFF design. In contrast, SR latches, such as the one depicted in Figure 4 are intrinsically symmetrical, making it more suitable to function as an unbiased arbiter.

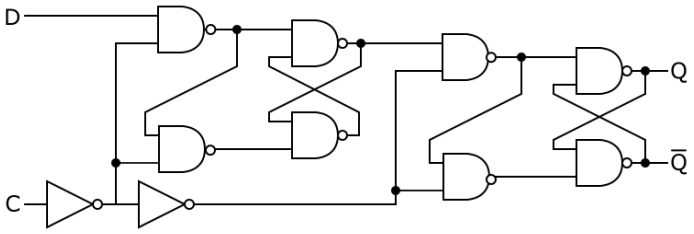

Figure 3: An example DFF design [1].

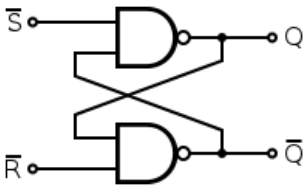

Figure 4: An SR latch [1].

Furthermore, if a trigger signal arrives at both the S and the R input ports at exactly the same time, then the output of the arbiter solely depends on the relative speeds of the two NAND gates. This is an important aspect for FPGA platforms, as identical logic paths are almost impossible to

come by. Often the minor speed differences at the transistor level are trumped by the enormous routing delay differences. Therefore, an SR latch can serve both as a good arbiter and an additional delay racing element. We will refer to this design as the SR arbiter from here on.

3.1.3.1 Implementation

We would like to highlight the necessity of nearly identical logic paths in the implementation of arbiters. Many obstacles, such as unpredictable cell placement and routing, are to be tackled to ensure that the competing paths to the arbiters are as symmetrical as possible.

Unfortunately, the target platform (Xilinx Virtex 5) does not offer any native SR latch logic cells. Therefore, an SR latch must be meticulously implemented by instantiating two lookup table (LUT) cells occupying the same logic slice (as NAND gates) and connecting them in combinational loops. By constraining the two LUT cells to the same logic slice, the combinational loop routing between the two LUTs are kept minimum and as close as possible.

Though the two NAND gates are identical in logic design, within each NAND gate, the two input-to-output paths are purposely designed to be different (as a matter of fact, the two paths cannot be designed to be identical due to the nature of SRAM-based LUT cells). While the Q-to-Qn and Qn-to-Q paths utilize the fastest path in each LUT (the highest address bit of the LUT), the S-to-Q and R-to-Qn paths utilize the slowest path in each LUT (the lowest address bit of the LUT). On the Virtex 5 FPGA where 6-input LUTs are available, the Q-Qn path only has one multiplexer, while the S/R-to-Q/Qn path has 6 layers of multiplexers. This arrangement echoes the desire to use the SR latch to measure the relative speed of the NAND gates.

To further ensure that the signal transitions arrive as close as possible at the S and R input ports, strict relative placement constraints are used to enforce an in-slice floorplan as illustrated in Figure 5. The two DFFs in the middle of the slice (B-DFF and C-DFF) store the Q and Qn results from the immediate NAND gates at B-LUT and C-LUT. The outer two DFFs (A-DFF and D-DFF) have very little clock skew between them, so they serve as precision triggers for the SR arbiter. To preserve local routing channels, the spare LUTs (A-LUT and D-LUT) are not allowed to be occupied by other functions. As a result, each SR arbiter occupies precisely one slice.

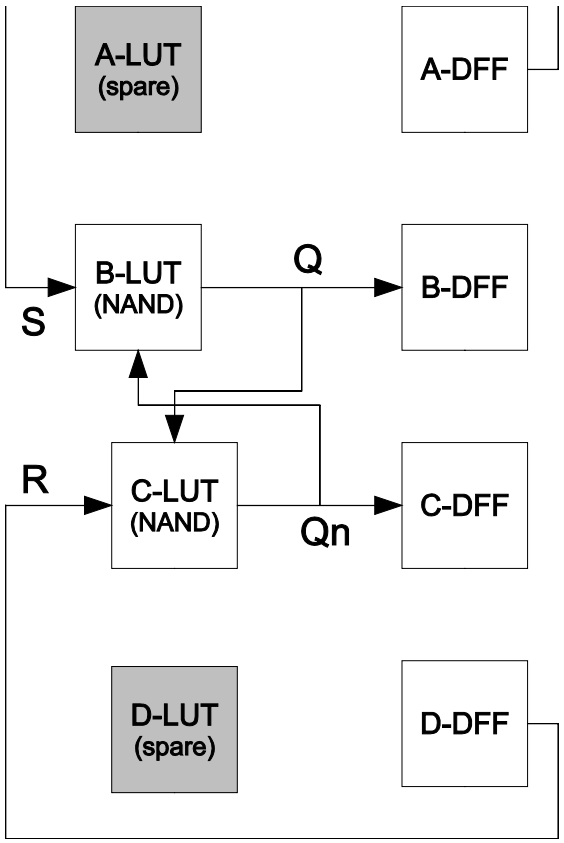

Figure 5: SR latch mapped to Virtex 5 CLB slice.

3.2 A Benign Hardware Trojan on FPGA-based Embedded Systems

Hardware Trojan Horses (HTHs) are hidden structural and functional alterations of an integrated circuit in such a way that the integrity or the privacy of the stored information is compromised. A favorite fictional example of the authors is an HTH-infested Ethernet interface card that can secretly transmit sensitive information to remote servers.

We discuss a less malicious type of HTH, whose intent is not to penetrate security walls, but to ward off security attacks on embedded systems, such as code-injection, reverse-engineering, and cloning. To be clear, the embedded system under consideration consists of a software component and a hardware component where the software is executed. The Benign Hardware Trojan (BHT), when embedded in the hardware component of the system, acts as a gate keeper to guarantee the following properties:

- That the software binary S_1 executing on the hardware platform H_1 is authorized to execute on H_1.

- That the hardware platform H_1 is an authorized execution target of the software binary S_1.
- That violating either of the previous two properties will cause the system to crash or stall.

3.2.1 Shared Uniqueness

Intuitively, the BHT is analogous to the unique genetic markings of a biological host waiting for a transplant. Only when the host (H_1) and the transplant (S_1) contain matching genetic markings can the transplant proceed. A key concept of the BHT implementation, therefore, is the embodiment of a shared uniqueness between H_1 and S_1; i.e. the software binary shall be tailored to only execute properly when the hardware manifests the shared uniqueness.

We propose to provide this uniqueness by exploiting process variation and aging effects from negative-bias temperature instability (NBTI) [50]. Using delay-sensitive logic, whose output behaviors are dependent on not only the structure of the logic circuits but also the relative speeds of competing logic paths [51], the minuscule intra- and inter-die delay variations can be translated to control signals whose outputs cannot be predicted or preset prior to manufacturing. In addition to delay variations that are introduced by process variations, selective aging can also be used to increase the delay disparities in the BHT.

Since the uniqueness of the hardware platform is a manifestation of a random physical process, reproducing it is extremely difficult, and the difficulty of reproduction increases exponentially as the size of the BHT logic increases.

Figure 6 demonstrates the feasibility of extracting unique silicon signatures based on process variation. In this figure, two structurally identical FPGAs (Xilinx Spartan-6) are configured using the same logic configuration, consisting of an array of 64 delay-sensitive arbiters. The outputs of the arbiters are dependent on the relative logic speeds of two competing paths, i.e. the output is zero if one path wins, and one if the other paths wins. The y-axis shows the accumulated output values of each arbiter over 1024 iterations. Roughly 30% of the arbiter outputs are different between the two “identical” FPGAs.

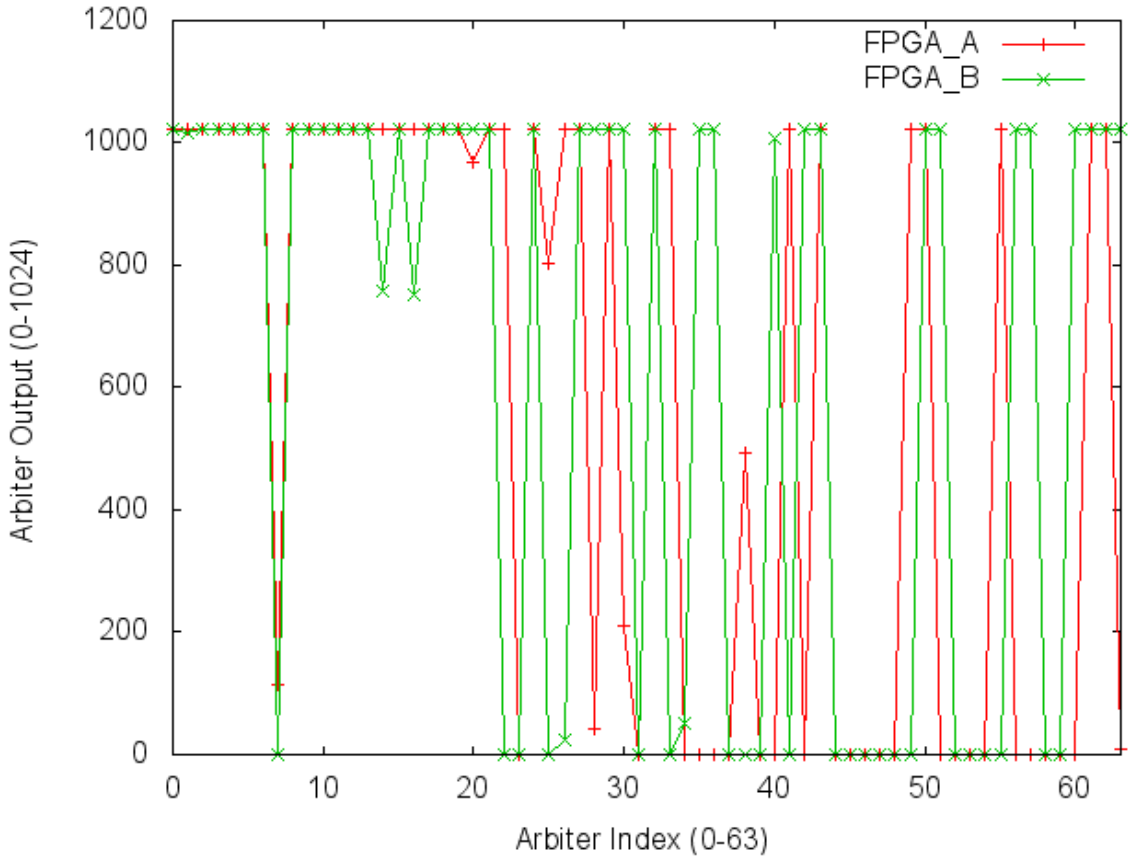

Figure 6: Comparison of arbiter outputs from two FPGAs.

3.2.2 Multi-Core Processor Example

In this section, we show a practical and intuitive example of incorporating the hardware signature with the software compilation process, which allows the software and the hardware to authorize each other during execution. This example is a system with a tiled multi-core processor (e.g. a tiled multi-core DSP processor) with as many as hundreds of processing cores. The highly optimized software implements a massively parallel algorithm with a *static* schedule of the processor cores.

We then use process variation to randomly choose *N* cores to be logically disabled by the BHT in each die. When the software is compiled, the states of the disabled cores are passed to the compiler to schedule the execution in such a way that the real computation only occurs on the enabled cores. The compiler can still schedule operations on the disabled cores, as long as the results do not contribute to the final answer.

We now analyze several possible reverse engineering or cloning attacks at the system and the defense options against them.

Let us assume that both the software and the hardware implementations are highly sophisticated, and therefore only low-level reverse engineering is possible. For the software component, it is possible to disassemble the binary code, but nearly impossible to obtain a high-level description of the software design. For the multi-core processor, we assume that an attacker can reverse-engineer the chip layout and routing or the configuration bitstream for FPGA designs to generate a low-level netlist. However, reverting the netlist to a behavioral description of the processor is also nearly impossible.

The first attack option is to attempt to clone the hardware system in such a way that all the processing cores are enabled after manufacturing. This attack shall be known as the functional clone attack. The functional clone is very difficult because a low-level cloning typically requires the clone to be as close to the original as possible. If the attacker makes a clone that's structurally identical to the original, process variation will guarantee that some cores will always be disabled by the BHT after manufacturing. Furthermore, the software binary can also include checks that verify the states of the cores that are meant to be disabled.

A second option is to manufacture a massive number of the hardware clones and hope to find one that will work with the cloned software code. This attack shall be known as the mining attack. The mining attack is prohibitively expensive, as the number of core choices grows binomially with the total number of cores. Figure 7 shows the growth of choices on a log scale with 16, 24, and 32 total number of cores. When half of the cores are disabled by the BHT, the growth of total choices is close to exponential growth.

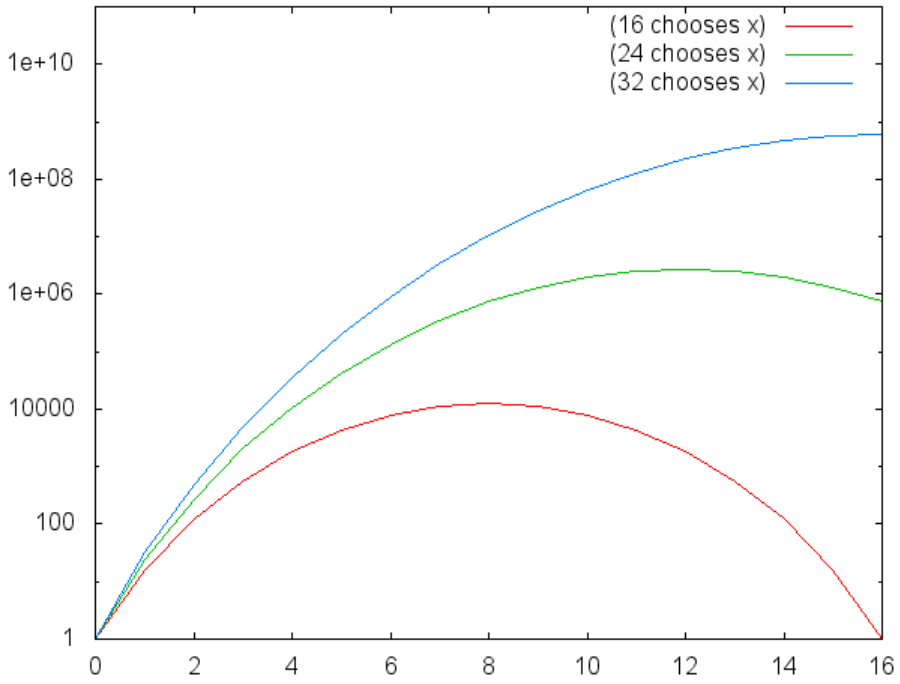

Figure 7: Choices for disabling cores when total number of cores is 16, 24, or 32.

A third option is to perform statistical analysis of the software binary's scheduling behavior to infer which processor cores are used and which ones are not. The attacker can then clone the hardware, map out the usable and unusable cores with the cloned hardware, and then patch up the static schedule in the software binary to use only the usable cores. This attack shall be known as the static analysis attack. The static analysis attack can also be guarded against if the compiler obfuscates the binary by scheduling both the functional and broken cores to active duty, but discard the outputs of the broken cores.

The final attack, dynamic analysis attack, is an extension to the static analysis attack. In a dynamic analysis attack, the attacker can write special test software to identify the disabled processing cores in the original processor, and use NBTI aging to make the clones behave like the original. The defense against the dynamic analysis attack, incidentally, is also NBTI aging. The BHT can be designed in such a way that its behavior and state is software-dependent. When the matched software executes on the hardware platform, it enforces an aging pattern which modifies the output of the BHT. Without understanding the aging profile applied by the original software, the special test software cannot learn the true state of the BHT.

3.2.3 Related Works

In this section we briefly survey directly related efforts, ranging from enabling technologies and techniques such as process variation, device aging, and hardware Trojans to already existing techniques for the protection of hardware and software.

Process variation is widely acknowledged as one of the most important challenges in modern designs [52]. Synthesis for FPGA considering process variation is an important research task that has been well addressed [53] [54]. Negative-Bias Temperature Instability (NBTI) is an intrinsic property of CMOS technologies, where each transistor under negative V_{gs} bias is subject to stress. A main ramification of NBTI is a significant speed degradation of the pertinent gates [50] [55]. Employment of device aging has been proposed for software metering [56], hardware-based cryptography [57], and prevention of reverse engineering [51].

Hardware Trojan has attracted a great deal of attention in the last five years [58] [59]. Some of the more recent detection techniques do not only detect the presence of hardware Trojans, but also enable their diagnosis, masking, or elimination [60] [61] [62] [63] [64] [65]. There have been a number of efforts to design small and/or highly damaging hardware Trojans [66] [67] [68]. However, while all these efforts created Trojan horses with the intention to demonstrate their practical importance and/or to facilitate development of the detection techniques, in our case the purpose of the Trojan horse is to directly enable the creation of new software and hardware protection techniques.

Several techniques for the protection of hardware intellectual property have been proposed. They can be classified into active and passive techniques. Active techniques have a high hardware overhead for remote activation and disabling [3]. Passive metering techniques have low cost overhead but they do not provide enforcement mechanisms [24]. Dabiri et al. proposed the addition of a high area and energy specialty circuitry for remote chip enabling and disabling and software usage metering [56].

To the best of our knowledge the new approach is the first software control that can used on both programmable processor and FPGAs. It does not require any additional circuitry when the platform is an FPGA because all the required device aging can be done using an FPGA configuration, which is consequently replaced with actual functionality.

4 RESULTS AND DISCUSSION

4.1 Securing Netlist-level FPGA Designs

The experimental setup and results of the SR-latch based arbiters are described in this section.

4.1.1 Experiment Setup

The main experiment of this paper carries two objectives. The first objective is to determine whether process variations, manifested as differences in propagation delay between two FPGA chips, can be effectively detected by SR arbiters. To achieve this, two SR arbiters with identical placement and routing are configured on two separate FPGA chips, and a trigger pulse is sent to the S and R ports of the arbiters. The output of the arbiter indicates whether the S path is faster than the R path, or the contrary. If the S path is consistently faster than the R path on both FPGAs, then the results will agree. However, if S path is faster than the R path on one FPGA, but slower on the other, then the arbiter results will disagree. Thus, by comparing the arbiter results of two FPGAs, the propagation delay differences can be detected.

The second objective is to determine whether the effects of NBTI aging and recovery can be detected by the SR arbiters. We will exploit the frequency dependency of the NBTI aging effect by maintaining the S and R inputs of the arbiter at either logic one or zero for a prolonged period of time in the hope to slow down the S or R path enough to change the outcome of the race between the S and R paths. Following the aging process, the static S and R inputs are changed to toggling between one and zero in order to recover from the aging effect. The output of the SR arbiter is measured and compared after each aging and recovery cycle.

4.1.1.1 Environment

The target platform used in this experiment is the Xilinx ML505 reference design board. The ML505 board is equipped with a Virtex 5 FPGA (v5lx50) that can be configured via a JTAG port. For the process variation objective, two such ML505 boards are used. The two FPGAs installed on the ML505 boards will be referred to as “FPGA-A” and “FPGA-B” from here on.

A desktop PC is used to configure and collect the results from the ML505 boards via RS-232 serial ports. All tests are conducted at ambient room temperature.

4.1.1.2 Baseline 64-Arbiter Array Design

To facilitate the test objectives of the experiment, an FPGA design populated with an array of 64 SR arbiters is implemented. Figure 8 shows a functional diagram of the arbiter array design. Besides the array of the SR arbiters, a timing unit is used to generate the trigger pulses to the array of arbiters at a rate of 160Hz. The results of the arbiters are then captured and transmitted by a UART encoder. A chip-level floorplan of the FPGA design is shown in Figure 9.

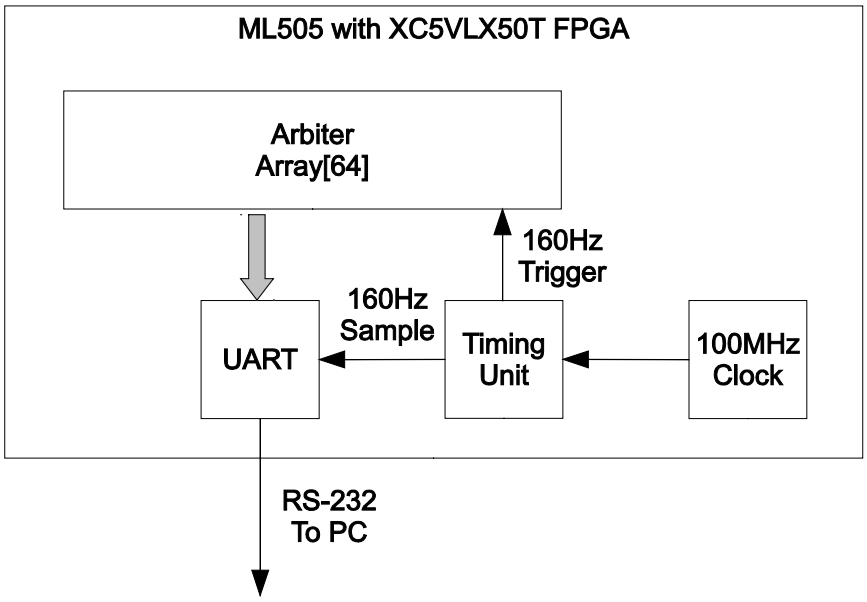

Figure 8: 64-Arbiter array test setup.

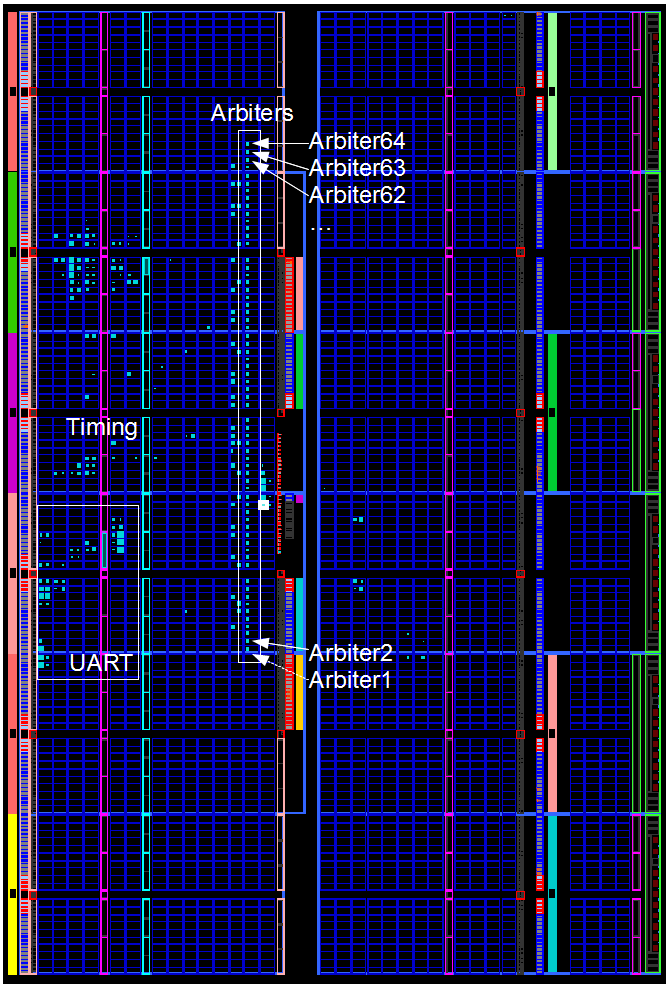

Figure 9: 64-Arbiter array floorplan.

4.1.1.3 Arbiter Result Scoring

An arbiter race result is represented in a binary format. A one indicates that the S path of the arbiter has won the previous race, while a zero indicates that the R path of the arbiter has won. Each result set contains 64 such binary values. A total of 1000 such result sets are collected by the host PC to compute an average score for each of the arbiters, i.e. a score of 0.0 means that the S path of the arbiter has won the race 1000 times, and a score of 0.6 means that the S path has won 600 times while the R path has won 400 times, etc. This average score is a reflection of the expected output of the arbiter under the same condition, and will be referred to as the "arbiter score" from here on.

4.1.2 Results

4.1.2.1 Process Variation

The first objective of the experiment is to determine whether process variation between two FPGAs can be detected by the SR arbiters. The results are collected before the first NBTI aging cycle and reflect the starting state of the FPGAs. Figure 10 plots the arbiter scores of the two FPGAs in a polar form, with the solid lines corresponding to FPGA-A, and the dashed lines corresponding to FPGA-B. The radius of each point on the plot reflects the arbiter score. For aesthetic reasons the inner circle represents a score of 0.0, and the maximum score (the tips of the longest spikes) is 1.0.

It is evident from the lack of overlaps between the solid and dashed lines in Figure 10 that the arbiter scores of the two FPGAs are significantly different. Since the two FPGAs are configured from an identical bitstream file, and the two FPGAs are exposed to the same ambient environment, we conclude that the differences in the arbiter scores observed at the same arbiter site are due to process variations at the transistor level.

Though it is possible that the minor differences in voltage and junction temperatures may contribute to the differences in arbiter scores, we believe that such differences will only cause a systematic shift of the scores. The seemingly random nature of the arbiter scores is more consistent with the effects of process variation.

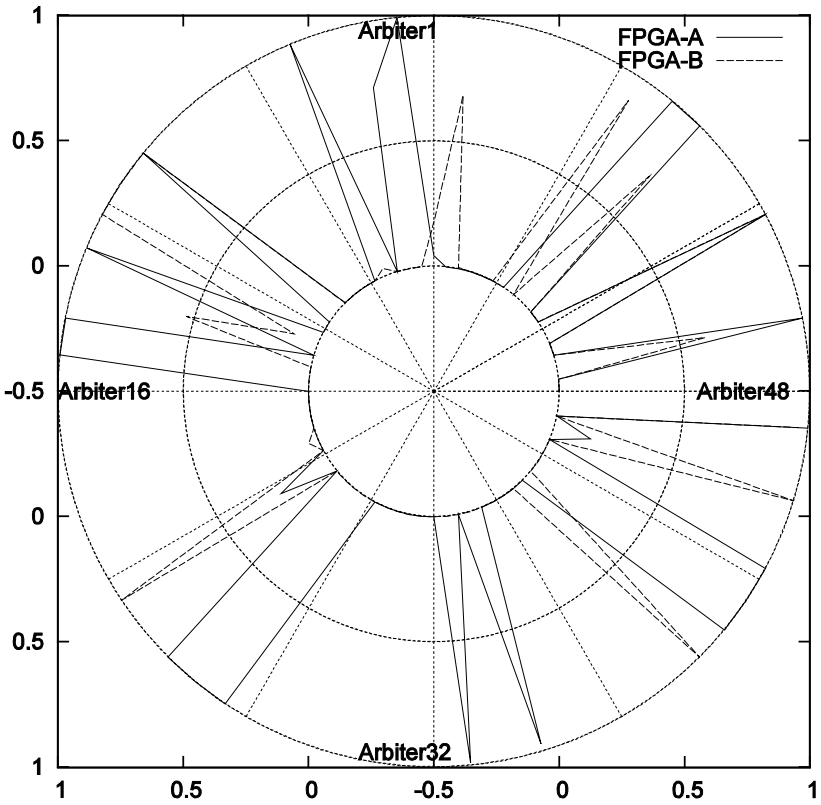

Figure 10: Arbiter score comparison.

4.1.2.2 NBTI Aging and Recovery

The second objective of the experiment is to determine whether NBTI aging and recovery effects can be detected by the SR arbiters. For this purpose, two variations of the baseline design are created. The first variation of the design applies a static logic pattern at the S and R ports of all arbiters to stress the S paths of the arbiters, while the second variation stresses the R paths. During an aging session, FPGA-A is treated with the first variation (stressing S paths), and FPGA-B is treated with the second variation (stressing R paths).

Each aging session lasts approximately 14 hours (overnight), followed by one or two days of recovery session where the baseline design is loaded and the S and R ports are constantly toggling. The arbiter scores are recorded between aging and recovery sessions.

In Figure 11 and Figure 12 results from two aging and one recovery sessions are presented. The y (vertical) axis reflects the arbiter scores for each arbiter (x-axis). The z-axis is a series of sample times in chronological order. In Figure 11 the arbiter scores for some arbiters in FPGA-A are pushed towards 0.0 after each aging session. This is consistent with the fact that the S paths

are being stressed during the aging session, and that an arbiter score closer to 0.0 reflects the higher likelihood of the R paths winning a race. After a recovery session, the arbiter scores move back towards 1.0, indicating that the NBTI stress on the S paths has receded, and that the S paths are becoming more likely to beat the R paths in a race.

The complete opposite takes place in Figure 12, where the R paths of the FPGA-B are stressed during the aging session. Note that on neither FPGA-A or FPGA-B did all arbiter scores change; in other words, the effect of a 14-hour continuous NBTI stress is not enough to change the outcome of the race between the S and R paths.

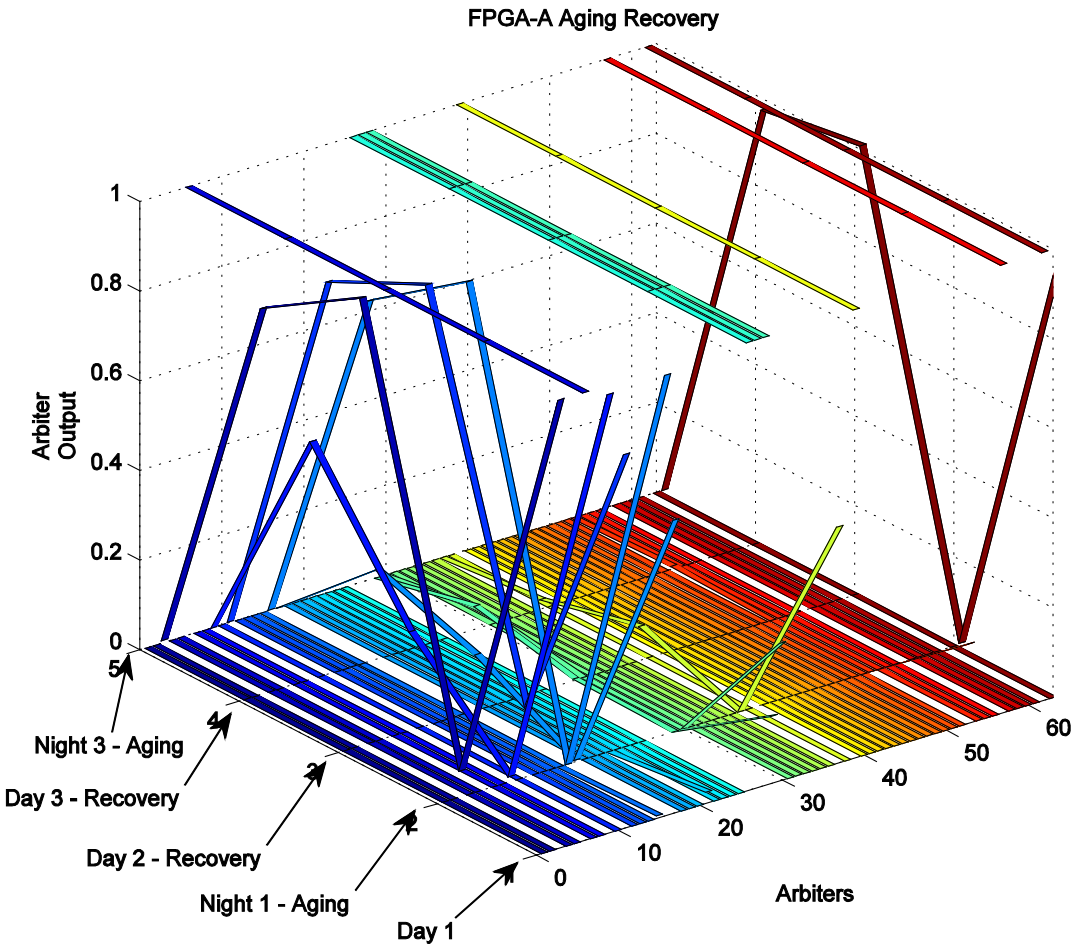

Figure 11: Aging and recovery of FPGA-A.

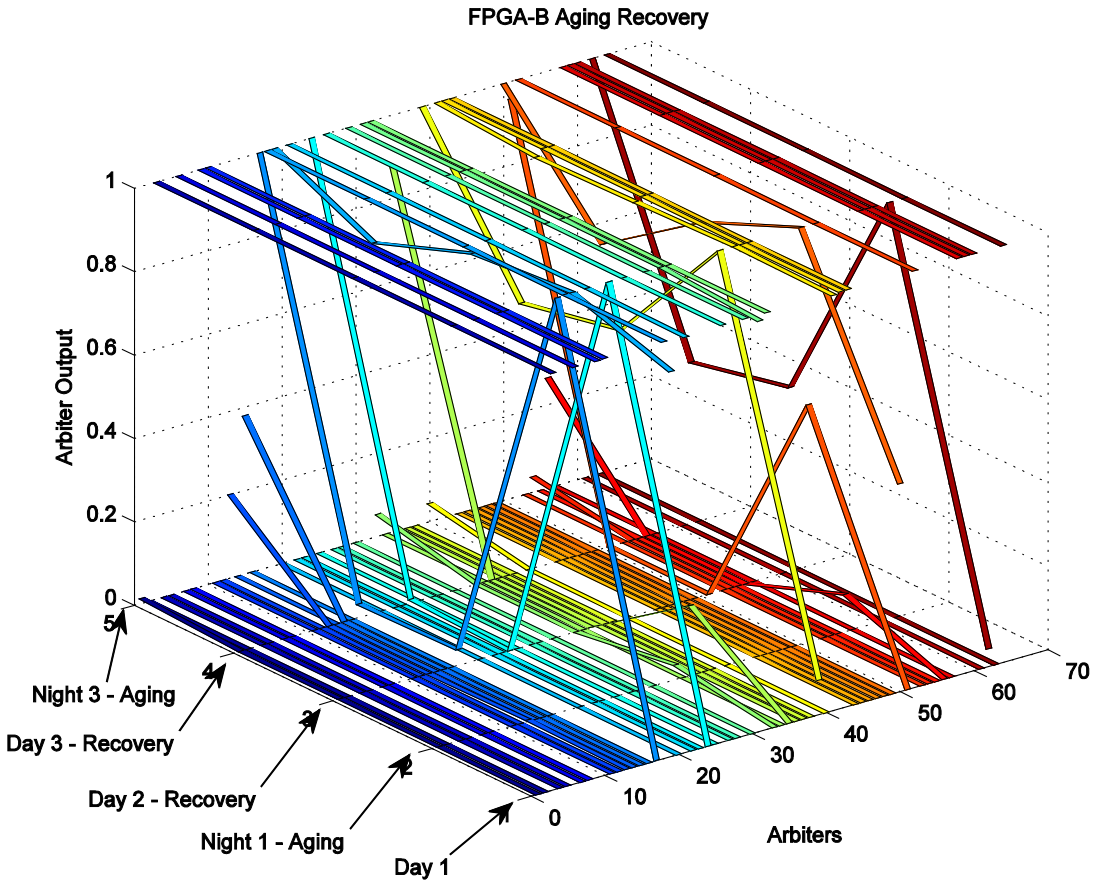

Figure 12: Aging and recovery of FPGA-B.

4.1.3 Case Study: LEON3 Processor

The LEON3 [72] is a 32-bit general-purpose processor based on the SPARC V8 architecture. The complete VHDL source code of the LEON3 is released under the GNU Public License for academic use. The LEON3 design is moderate in size and sophistication, therefore we chose to use the LEON3 as a platform to demonstrate how arbiter delay logic can be easily incorporated into a modern design to prevent unauthorized cloning and reverse engineering. There are many techniques for delay logic to be incorporated. In this paper, we demonstrate the simplest method, which is to use arbiter outputs as a unique ID to control a shuffling network embedded in the control logic paths of the target design.

4.1.3.1 Control Logic Shuffling

A simple way to implement anti-cloning is to use physical signatures of the device to scramble control signals in a controlled way. As an analogy to cryptography, the control signals

are like messages that can be encrypted, transmitted, and decrypted to be used. The encryption key is the physical signature of the chip; in this case, the arbiter scores and the decryption key are functions of the encryption key.

One method to scramble control signals is to shuffle logic. A two-input shuffler has two data inputs, two data outputs, and a control input. When the control input is set to logic zero, the two data outputs are exact copies of the two data inputs: data input A drives data output A, and data input B drives data output B. When the control input is set to logic one, the output orders are swapped: data input A drives data output B, and data input B drives data output A. A tree of such shuffling logic controlled by arbiter outputs can ensure that the design will only function properly on FPGA devices with a matching arbiter output signature.

4.1.3.2 Selection of Control Logic

At the core of the LEON3 processor is a seven-stage (fetch, decode, register read, execute, memory, exception, and register write) integer unit. Since all instructions must go through the integer unit, the control logic in the pipeline stages is an ideal place to employ the shuffler technique to prevent unauthorized design cloning.

To minimize the performance impact on the LEON3, the selection of the control logic for shuffling is made after first sorting the control logic paths by timing slack. The control logic with the largest amount of setup slack is least likely to become the critical path after shuffling logic is inserted. Evidently the ALU control signals (ALUOP) have the most slack among all control signals in the integer unit after examining the static timing analysis. What is also interesting about the ALU control signals is that they are highly critical to the normal operation of a processor. Therefore the ALU control signals are chosen as the target of the shuffling logic.

4.1.3.3 Static Shuffling Strategy

The basic strategy to organize static shuffling is described in this section. The strategy can be summarized in three words: shuffle, rotate, and reorder. These three words represent the three types of operations that can be performed on a group of control signals.

The shuffle operation mainly makes use of two-input shufflers. Each shuffler, depending on the control port ShuffleEn (SE), can either swap the two input signals or do nothing to them. Shown as the first and third stages in Figure 13, two adjacent control signals are sent to a shuffler. The ShuffleEn signal is controlled by the output of an SR arbiter. If there is an odd

number of control signals to shuffle, the last signal is sent to a shuffler with its inverted version, i.e. the shuffler can either send out the original or the inverted version of the signal, depending on the state of the ShuffleEn port. This type of shuffler is also referred to as an Inv-Shuffler.

The rotate operation always performs a single-bit rotation on the group of input signals. The direction of the rotation is decided by the RDIR input port, which is also driven by an SR arbiter.

The shuffle and rotation operations can be repeated as many times as possible, so long as the timing slack is sufficient. Figure 13 shows the shuffle and rotate interleaving each other to incrementally introduce entropy into the system.

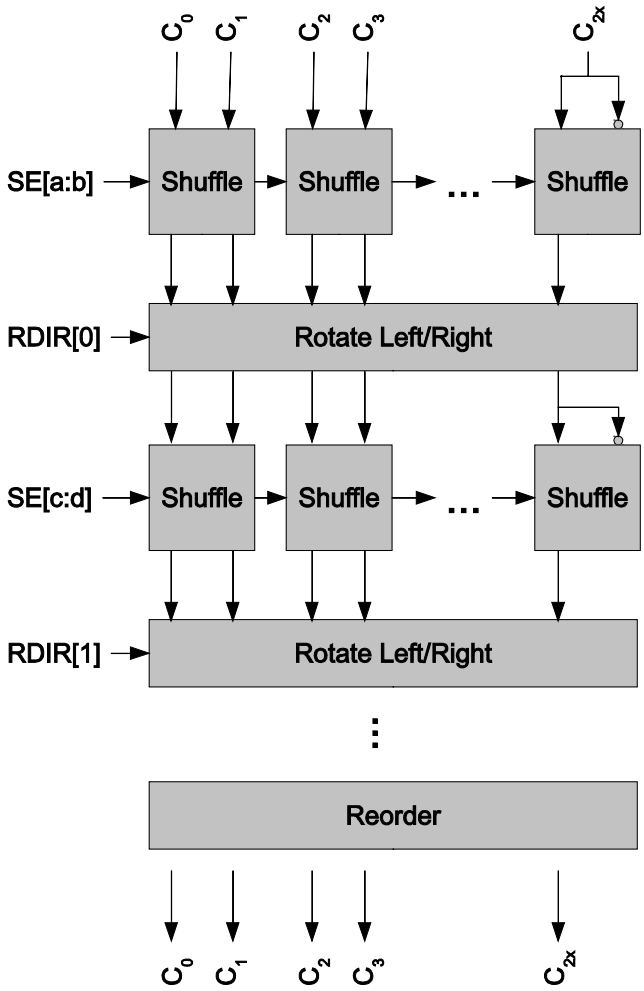

Figure 13: Shuffle, rotate, and reorder.

The reorder operation is always the last stage before the control signals leave the shuffling tree. The purpose of having the reorder operation is to maintain the original order and polarity of

the signals the same as they enter the shuffling tree. For example, if signal A and signal B have their positions swapped by a shuffler, then the reorder stage must swap them back.

In a fully static shuffling tree, all the SE and RDIR inputs are controlled directly by the outputs of SR arbiters. At compile/synthesis time, the reorder stage must know ahead of time what the SR arbiters will output to properly reorder the signals.

4.1.3.4 Metrics

The implementation result of the LEON3 with static shuffling will be judged on various metrics. The first metric is the functional correctness of the processor. The modified LEON3 processor must remain functional as before on the intended FPGA target, and any unauthorized copy must render the processor unusable. This is tested by loading the same LEON3 design on two ML505 boards. Only one of the ML505 board has the FPGA that matches the SR arbiter outputs expected by the reorder stage, i.e. this is the intended FPGA target. The other ML505 board, when loaded with the same design, should not function properly as a processor.

The second metric is the amount of area and performance overhead incurred by the modification. The area overhead is measured in number of LUTs used, and the performance overhead is measured by the minimum clock cycle, or the maximum clock speed.

The last metric is the most difficult to quantify: how easy can the anti-cloning scheme be attacked? To answer this question, we must make several assumptions about the attacks:

- The bitstream of the design can be recovered from the EEPROM storage.
- With the proper know-how, the bitstream can be converted into a netlist.
- The SR arbiters can be readily identified from the netlist. However, the outputs of the SR arbiters cannot be determined statically, nor can they be measured dynamically.
- The attacker therefore must resort to a brute force attack by guessing the outputs of the arbiters and verify the correctness of their guesses by running a netlist-level simulation.

Using the above assumptions, the difficulty of the attack then depends exponentially on the number of arbiters used, and linearly on the number of simulation cycles required to verify the correctness of the guess. Since the number of arbiters is limited and trivial to determine, we decided to use the number of simulation cycles to verify correctness as the metric against attacks.

4.1.3.5 Results

Figure 14 shows the floorplan of the modified LEON3 processor implemented on the v5lx50 FPGA. In the picture, DIV refers to the radix-2 integer divider, MUL refers to the dedicated multiplier. The integer unit, shown in orange, occupies most of the north side of the FPGA. The ALU control shufflers occupy a very small area (inside the white circle). A magnified view of the shufflers can be seen in Figure 15, where the LUTs implementing the shuffle and rotate stages are pointed out. As a proof of concept, only three arbiters are used to statically control the shuffle and rotation stages. The three DFFs (highlighted in white in Figure 15) store the arbiter outputs from the selected arbiters. The two DFFs on the left are expected to hold a value of one, and the DFF on the right is expected to hold a value of zero in order for the processor to function properly.

To answer the metric of functional correctness, the modified LEON3 design is loaded on two ML505 boards. A test program that finds all prime numbers less than 1000 is then loaded to the processor to run. On the FPGA with the correct arbiter outputs, the results are returned in exactly the same way as an unmodified LEON3 processor design would. On the FPGA with the incorrect arbiter outputs, the computation did not complete. Instead, the processor quickly traps to an error state. This proves that the modified LEON3 design passes the functional correctness metric.

To evaluate the area and performance overhead of the static shuffling and arbiter array, the FPGA mapping and static timing reports are examined and compared against the unmodified version of the LEON3 design. As shown in Table 4, the static shufflers and the arbiter array has a very small resource overhead, using only 0.4% additional LUTs and 6.3% additional DFFs. There is no timing overhead. In fact the modified LEON3 runs slightly faster than the unmodified LEON3.

The last metric is the number of simulation cycles the design must be simulated before an attacker can determine that the guessed arbiter outputs are incorrect. By simulating an instance of the modified LEON3 design with incorrect arbiter outputs until the process traps to an error state, the answer is determined to be 6.2585 billion simulation cycles.

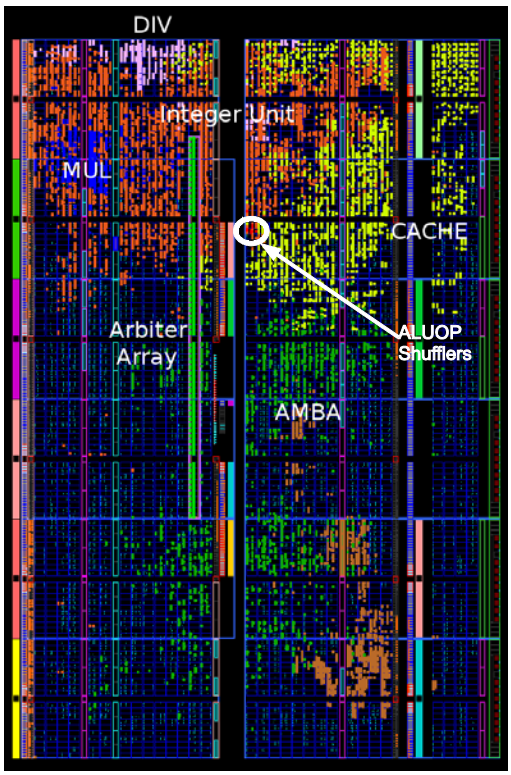

Figure 14: LEON3 floorplan.

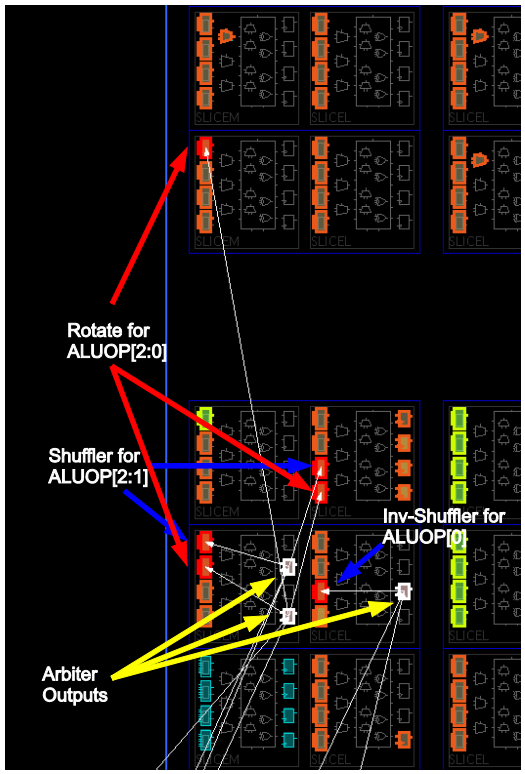

Figure 15: ALUOP shufflers.

Table 3: Area and performance overhead.

	Unmodified LEON3	Modified LEON3	Overhead
LUT Usage	15,271	15,334	0.4%
DFF Usage	7,649	8,132	6.3%
Clock Speed	80.18MHz	80.48MHz	-0.4%

4.2 BHT on FPGA-based Embedded Systems

4.2.1 Implementation Approach

The multi-core processor case in Section 3.2.2 serves as an intuitive example of BHT, and it would have been an impressive demonstration if implemented. However, due to the limited reconfigurable logic resources available at our disposal, we choose to demonstrate the BHT with a simpler form of commodity resources in an embedded system. We would like to show that BHT can be used to control the disabling of the general purpose registers (GPRs) on an embedded microprocessor with a low performance overhead. With the latency of a single cycle, GPRs are the processor's fastest and most frequently used data storage unit. Similar to the multi-core processor example, there are usually more than enough GPRs on a modern processor architecture, and any typical modern compiler can handle register scheduling with an arbitrary set of GPRs.

Following the example of the multi-core processors, we incorporate the BHT into the processor to control the disabling of GPRs. The modified processor is synthesized, placed, and routed on an FPGA platform to run. The outputs of the BHT control path are characterized dynamically when the processor is running. With a BHT array directly controlling which GPRs are enabled, both the functional clone attack and the mining attack are deterred.

On the compiler side, the compiler is modified such that the knowledge of which GPRs are not available is to be passed on at the compile time. The compiler then generates a binary code that only uses the GPRs that are available. Finally, the compiled code is executed on the physical processor on the FPGA board for benchmarking.

To guard against the static and the dynamic analysis attacks, the compiler requires further modification to its scheduling algorithm and additional NBTI characterization on the selected target device to show the effects of NBTI on the BHTs (a similar characterization was performed on Virtex 5 FPGAs in [51]). These two attacks will be addressed in a future implementation.

4.2.1.1 Hardware Implementation

We choose to implement the BHT on an open-sourced processor. The OpenRISC OR1200 is a five-stage, 32-bit general-purpose processor. There are 32 general-purpose registers in the architecture, and most of which can be masked without functional impact to the processor (exceptions are covered in Section 4.2.1.2). We choose the OpenRISC platform because the complete set of source code is available (in Verilog format), which is critical to modifying the processor design.

The BHT is implemented as delay-logic arbiters as described in [51], whose outputs enable or disable writing to particular GPRs, as depicted in Figure 16. Each arbiter output corresponds to a particular GPR number. When a disabled GPR is decoded as the destination register for an instruction, the output of the BHT masks the write, thus effectively disabling the register.

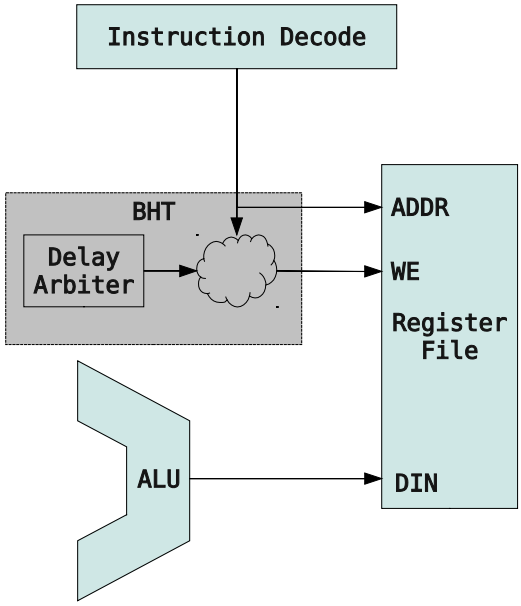

Figure 16: BHT masking selected GPR writes.

The Digilent Atlys board from the Xilinx University Program is used as the target platform, which has a 45nm-process Spartan-6 FPGA and 128 megabytes of DDR2 memory. The OR1200 processor is implemented with a full suite of on-chip peripherals, including a DDR2 memory controller, Ethernet controller, serial communication, hardware-based multiplier, divider, and floating-point unit. The processor and its peripherals take up roughly 50% of the available logic resources, and is clocked at 50MHz. Table 4 shows a summary of the resources used in the FPGA, and Figure 17 shows the final layout of the design.

Table 4: OR1200 resource usage.

Resource	Use Count	Use Percentage
D-Flipflop	5,718	10%
LUTs	10,918	40%
Slices	3,661	53%
BRAMs	87	37.5%
DSP48A1s	4	6%

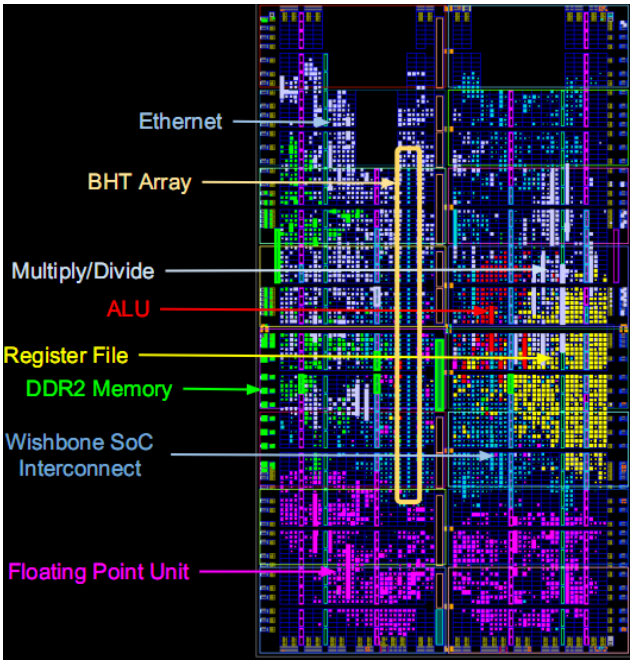

Figure 17: Layout of the implemented OR1200 with BHT.

4.2.1.2 Software Implementation

As an embedded platform, OpenRISC offers several options for building and running code. OpenRISC utilizes the common GNU toolchain for 32-bit C/C++ support, including GCC, binutils, and GDB. However, library support is provided by either Newlib, uClibc, or the low-level System-on-Chip (SoC) platform libraries. Newlib provides libc support for stand-alone bare-metal code, while uClibc provides libc support for Linux. The much more limited SoC platform libraries can also be utilized to run bare-metal code, although they provide only a small subset of the functionality available from the libc implementations. Although libc libraries like Newlib and uClibc are available, much functionality is still provided through OpenRISC-specific extensions, and not the standard functions. In addition to the GNU toolchain, the OpenRISC

platform provides additional debugging tools such as the adv_debug_bridge JTAG proxy and the or1ksim architectural simulator.

As we aim to evaluate the performance impact of reduced general purpose register availability, the software tool-chain of the OpenRISC platform must be modified to allow arbitrary sets of registers to be disabled. We do this by extending the options available in the GCC OpenRISC target to modify register availability. This functionality is available as a command line option under the -mregistermask flag, taking a 32-bit value that represents the state of each register as indexed by the bit positions, where the least significant bit represents GRR0. Setting the bit makes the corresponding GPR unavailable to the compiler.

Although the 32-bit OpenRISC architecture defines 32 general purpose registers, several are reserved for special purposes by the GCC OpenRISC machine definition and cannot be disabled. They are as follows:

- R0, defined as the zero constant by the 32-bit OpenRISC architecture,
- R1, defined as the stack pointer by the GCC,
- R9, defined as the link pointer by the GCC,
- R10, defined as the Linux thread register by the GCC.

In addition to the GPR modifications in the GCC, we modify the or1ksim architectural simulator to simulate disabled registers for testing purposes. In much the same way that the GCC flag -mregistermask works, the CPU configuration option disable_regs was implemented in the modified or1ksim to selectively disable registers using a 32-bit integer mask. By inserting code into the simulator state machine generated, register checks were added to trigger a processor exception upon accessing the disabled GPR. This allowed us to do preliminary characterization using the simulator's built-in cycle counter.

4.2.2 Benchmark Results

Due to the fact that the scheme relies on disabling registers, the performance impact must be considered. We have thus benchmarked the performance of the modified OpenRISC platform using a variety of programs, varying the numbers of disabled registers. As the OpenRISC platform is an embedded system with limited support for file access to storage, we relied primarily on self-contained embedded system benchmarks. Evaluations of register performance penalties were done using a small suite of benchmarks consisting of Dhrystone [70], CoreMark

[71], a subset of MiBench [72], and zlib [73]. Dhrystone and CoreMark are synthetic benchmarks that attempt to capture typical CPU workloads, while MiBench attempts to characterize CPU performance based on a variety of common operations, including FFT (Fast Fourier Transformation), CRC-32 (Circular Redundancy Checks), string searching, media decode and encode, and encryption.

Both CoreMark and Dhrystone are completely synthetic benchmarks, and not centered around real workloads, but instead seek to approximate real embedded workloads. Dhrystone has been an industry standard since the 1990s, and consists of a mix of assignments, pointer arithmetic, and procedure calls. CoreMark is created by the Embedded Microprocessor Benchmark Consortium (EEMBC) targeted at evaluating embedded systems. It consists of modules that evaluate linked list, matrix, and state machine performance, making it a more realistic simulation of common embedded workloads than Dhrystone.

MiBench was originally developed at the University of Michigan to evaluate embedded system performance as CoreMark was previously not freely available. It consists of 35 individual modules that are based on a variety of real-world workloads. As most of the MiBench modules rely on external data to operate, we are only able to make use of a subset of the modules, including the basic math, bitcount, FFT, and string search modules. The basic math module is designed to test math operations that are not commonly supported by hardware, such as integer square root, cubic function solving, radian to degree conversion, etc. The bitcount module is designed to evaluate the bit manipulation performance of an embedded processor, counting bits using a variety of methods. The FFT module performs a floating point Fast Fourier Transform on a polynomial function with pseudo-randomly generated amplitude and frequency components. Finally, the string search module evaluates performance using a fast string search to find strings in a large array of input phrases.

The zlib benchmark is naturally designed as a real-world test of compression performance, as the DEFLATE algorithm used in zlib is used very frequently in real software. The benchmark itself is very simple and centers around initializing large chunks of random data, followed by compressing those chunks using calls to deflate functions in zlib. Since the OpenRISC platform does not allow file access, input data is randomly generated and the resulting compressed data is merely thrown away, making this a test purely of compression speed without file I/O to slow things down.

4.2.2.1 Measurements Methodology

To measure the performance using these various benchmarks, we used extensions present in the OpenRISC Newlib implementation. These extensions provide access to the CPU timer, enabling accurate timing, as Newlib does not provide a functional implementation of standard libc timer code for the OpenRISC port. Timer accuracy is set to 1000Hz for all measurements, making the ticks measured equivalent to time in milliseconds. Each benchmark measures the elapsed time in number of ticks.

To see the impact of available registers on benchmark performance, the number of available GPRs varies with the following numbers: 32, 28, 24, 22, 20, 18, and 16. With each variation, the benchmark is recompiled with a corresponding register mask. Each test is repeated three times, and the lowest number is used as the result. The first run is typically slower due to cache misses.

As fewer registers are available, the processor is forced to visit the memory system more frequently. A special test is run to evaluate the cache dependence of the performance penalty. This test is only run with CoreMark, where the entire test is repeated with the cache disabled.

4.2.2.2 Performance Penalty

The raw benchmark results are tabulated in Table 5. Using the results from perspective 32-GPR runs, the benchmark numbers are normalized to a base number of 100, and the results are graphed in Figure 18.

Excluding the cache-disabled CoreMark runs, the worst case performance penalty is 8% for zlib with 22 GPRs. Most of the benchmarks show a monotonic increase in run time with the decrease of available GPRs. The Dhrystone result is an interesting outlier, where we observed the worst-case performance when all 32 GPRs are available. We are not completely surprised by this counter-intuitive behavior, as the Dhrystone benchmark performance has been shown to be highly dependent on compiler optimization [70].

A similar anomaly can be seen with the zlib results. The run time monotonically increases until the number of available GPRs is reduced to 20, where the run time drops to lower than the 32-GPR run, and then increases monotonically again. We suspect that this is also an artifact of the compiler optimization choices.

The basic math and FFT module results from MiBench show very little run time impact as a result of reduced general purpose register availability. This is likely due to the fact that basic math performance is limited by arithmetic hardware, rather than register availability, while FFT largely performs floating point operations that do not use the general purpose register set, and is limited by the Floating Point Unit's (FPU) latency.

The impact of constraining the number of GPRs in the absence of cache can be observed in Figure 19. The absence of cache compounds the performance penalty at every variation of GPR numbers. The effect is especially high when only 16 GPRs are available, resulting in a 20% increase in run time.

Table 5: Raw benchmark run times (1000 Hz ticks).

Benchmark	32 GPRs	28 GPRs	24 GPRs	22 GPRs	20 GPRs	18 GPRs	16 GPRs
MiBench.basicmath	39640	39641	39645	39659	39667	39675	39680
MiBench.fft	6570	6574	6580	6580	6674	6675	6578
MiBench.bitcnt	24325	24325	24325	24325	24483	24800	26068
MiBench.stringsearch	106	106	106	106	107	108	108
Dhrystone	4032	1633	1642	3641	3645	3642	3662
CoreMark	1563	1560	1571	1590	1592	1621	1633
CoreMark no_cache	2102	2113	2130	2145	2162	2210	2522

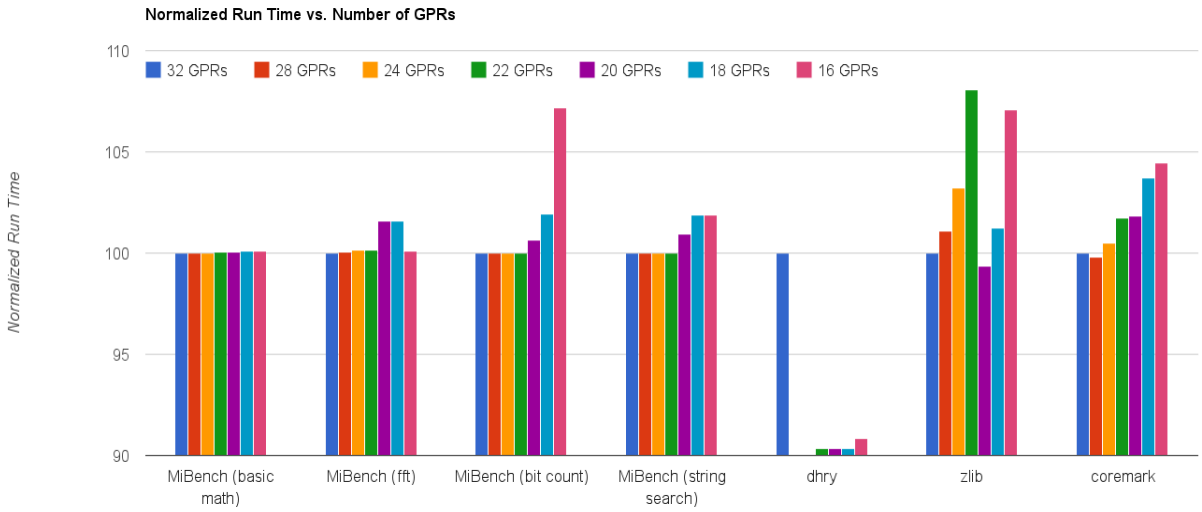

Figure 18: Performance penalties from fewer GPRs.

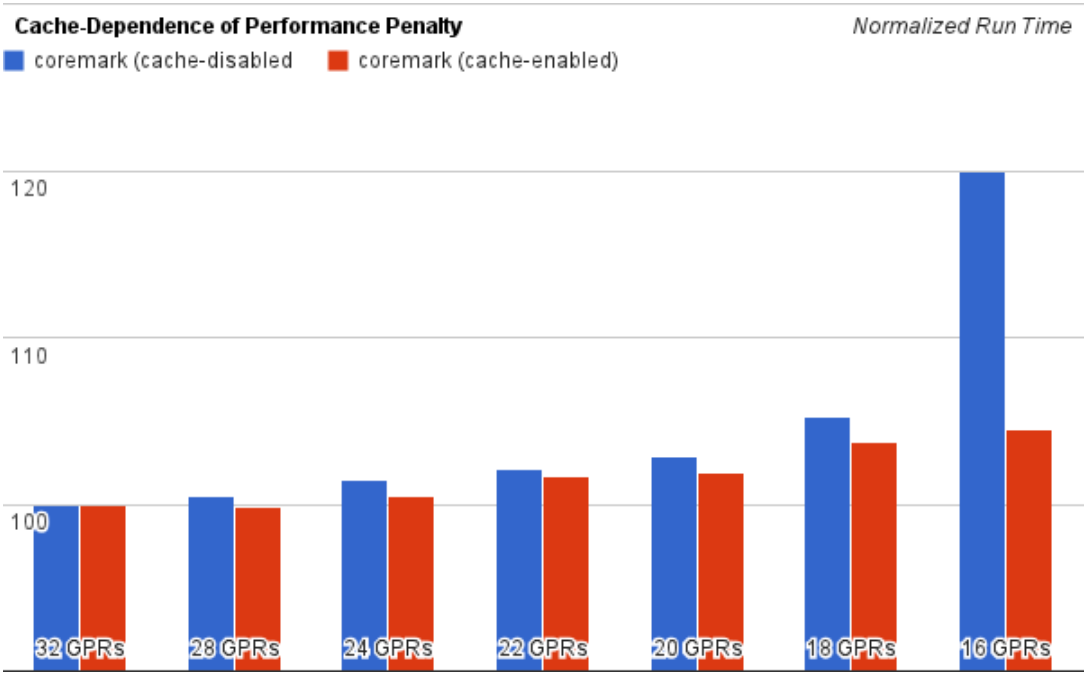

Figure 19: Performance penalty and cache.

5 CONCLUSIONS

We have introduced a new approach to IC Digital Rights Management by combining directed NBTI aging and delay logic. As a proof of concept, we have implemented a basic form of the delay logic on an FPGA platform, and shown that the delay logic measures the relative speed differences in competing logic paths due to process variation. Furthermore, we have shown that the delay logic responds correctly to NBTI aging and recovery cycles. We also presented a case study to use static arbiter outputs and shuffling logic to add anti-cloning protection to a non-trivial FPGA design. We showed that with very few resources and no clock speed overhead, the LEON3 processor design can be made clone-proof.

We have also presented the novel use of a benign hardware Trojan as a security measure when combined with delay logic, process variation, and NBTI aging. We have shown through example that the BHT can be very resilient against reverse-engineering and cloning attacks. Furthermore, we have demonstrated a successful implementation of an embedded system incorporated with BHT. Finally, we have shown that adding BHT as a security measure only brings about a modest amount of performance penalty when tested against a suite of benchmarks.

6 REFERENCES

[1] Wikipedia, "Flip-flop (electronics) – wikipedia, the free encyclopedia," 2011. [Online; accessed 26-September-2011].

[2] Y. Alkabani and F. Koushanfar, "Active hardware metering for intellectual property protection and security," USENIX Security, pp. 291-306, 2007.

[3] Y. Alkabani, F. Koushanfar, and M. Potkonjak, "Remote activation of ICs for piracy prevention and digital right management," ICCAD, pp. 674-677, 2007.

[4] J. Roy, F. Koushanfar, and I. Markov, "EPIC: Ending piracy of integrated circuits," DATE, pp. 1069-1074, 2008.

[5] A. Baumgarten, A. Tyagi, and J. Zambreno, "Preventing IC piracy using reconfigurable logic barriers," IEEE Design & Test of Computers, vol. 27, no. 1, pp. 66-75, 2010.

[6] T. Kean, "Method of using a mask programmed key to securely configure a field programmable gate array." United States Patent 7,240,218, 2001.

[7] L. Bossuet, G. Gogniat, and W. Burleson, "Dynamically configurable security for SRAM FPGA bitstreams," Proceedings of the Parallel and Distributed Processing Symposium, pp. 146-153, April 2004.

[8] S. Drimer, "Volatile FPGA design security - a survey (v0.96)," April 2008.

[9] M. A. Alam, "A critical examination of the mechanics of dynamic NBTI for PMOSFETs," IEDM, p. 346, 2003.

[10] S. Mahapatra, M. A. Alam, P. B. Kumar, T. R. Dalei, D. Varghese, and D. Saha, "Negative bias temperature instability in CMOS devices," Microelectronic engineering, vol. 80, pp. 114-121, 2005.

[11] S. Trimberger, "Trusted design in FPGAs," DAC, pp. 5-8, 2007.

[12] M. Majzoobi, F. Koushanfar, and M. Potkonjak, "Trusted design in FPGAs," Introduction to Hardware Security and Trust, pp. 195-230, Springer, 2011.

[13] J. Lach, W. Mangione-Smith, and M. Potkonjak, "FPGA fingerprinting techniques for protecting intellectual property," Custom Integrated Circuits Conference, pp. 299-302, 1998.

[14] J. Lach, W. Mangione-Smith, and M. Potkonjak, "Robust FPGA intellectual property protection through multiple small watermarks," DAC, pp. 831-836, 1999.

[15] J. Lach, W. Mangione-Smith, and M. Potkonjak, "Fingerprinting techniques for field programmable gate array intellectual property protection," IEEE Transactions on CAD, vol. 20, no. 10, pp. 1253-1261, 2001.

[16] D. Champagne, R. Elbaz, C. Gebotys, L. Torres, and R. B. Lee, "Forward-secure content distribution to reconfigurable hardware," Reconfigurable Computing and FPGAs, pp. 450-455, 2008.

[17] R. Anderson and M. Kuhn, "Tamper resistance: A cautionary note," Proceedings of the 2nd USENIX Workshop Electronic Commerce, pp. 1-11, 1996.

[18] M. Majzoobi, F. Koushanfar, and M. Potkonjak, “Techniques for design and implementation of secure reconfigurable PUFs,” ACM Transactions on Reconfigurable Technological Systems, vol. 2, pp. 5:1-5:33, March 2009.

[19] U. Ruhrmair, F. Sehnke, J. Solter, G. Dror, S. Devadas, and J. Schmidhuber, “Modeling attacks on physical unclonable functions,” ACM Conference on Computer and Communications Security, pp. 237-249, 2010.

[20] J. Lach, W. Mangione-Smith, and M. Potkonjak, “Fingerprinting digital circuits on programmable hardware,” Information Hiding Workshop, pp. 16-31, 1998.

[21] I. Hong and M. Potkonjak, “Technique for intellectual property protection of DSP designs,” International Conference on Acoustic, Speech, and Signal Processing, pp. 3133-3136, 1998.

[22] G. Qu and M. Potkonjak, Intellectual Property Protection in VLSI Design Theory and Practice. Springer, 2003.

[23] F. Koushanfar, G. Qu, and M. Potkonjak, “Intellectual property metering,” Information Hiding Workshop, pp. 81-95, 2001.

[24] Y. Alkabani, F. Koushanfar, N. Kiyavash, and M. Potkonjak, “Trusted integrated circuits: A nondestructive hidden characteristics extraction approach,” Information Hiding Workshop, pp. 102-117, 2008.

[25] S. Wei, F. Koushanfar, and M. Potkonjak, “Integrated circuit digital rights management techniques using physical level characterization,” ACM Workshop on Digital Rights Management, pp. 3-14, 2011.

[26] S. Wei, A. Nahapetian, and M. Potkonjak, “Robust passive hardware metering,” ICCAD, 2011.

[27] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas, “Silicon physical random functions,” ACM Conference on Computer and Communications Security, pp. 148-160, 2002.

[28] N. Beckmann and M. Potkonjak, “Hardware-based public-key cryptography with public physically unclonable functions,” Information Hiding Workshop, pp. 206-220, 2009.

[29] S. Blythe, B. Fraboni, S. Lall, H. Ahmed, and U. de Riu, “Layout reconstruction of complex silicon chips,” IEEE Journal of Solid-State Circuits, vol. 28, no. 2, pp. 138-145, 1993.

[30] R. Nakagaki, Y. Takagi, and K. Nakamae, “Automatic recognition of circuit patterns on semiconductor wafers from multiple scanning electron microscope images,” Measurement Science and Technology, vol. 21, no. 8, p. 085501, 2010.

[31] J. Kumagai, “Chip detectives,” IEEE Spectrum, vol. 37, no. 11, pp. 43-48, 2000.

[32] M. C. Hansen, H. Yalcin, and J. P. Hayes, “Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering,” IEEE Design & Test, vol. 16, no. 3, pp. 72-80, 1999.

[33] J. L. White, Candidate subcircuit enumeration for module identification in digital circuits. PhD thesis, Michigan State University, East Lansing, MI, 2000.

[34] J. L. White, A. S. Wojcik, M.-J. Chung, and T. E. Doom., "Candidate subcircuits for functional module identification in logic circuits," Great Lakes Symposium on VLSI, pp. 34-38, 2000.

[35] T. Doom, J. White, A. Wojcik, and G. Chisholm., "Identifying high-level components in combinational circuits," Great Lakes symposium on VLSI, pp. 313-318, 1998.

[36] J. D. Parham, J. T. McDonald, M. R. Grimaila, and Y. C. Kim, "A java based component identification tool for measuring the strength of circuit protections," Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research, pp. 26:1-26:4, 2010.

[37] J. T. McDonald, E. D. Trias, Y. C. Kim, and M. R. Grimaila, "Using logic-based reduction for adversarial component recovery," ACM Symposium on Applied Computing, pp. 1993-2000, 2010.

[38] J. T. McDonald, Y. C. Kim, and M. R. Grimaila, "Protecting reprogrammable hardware with polymorphic circuit variation," Cyberspace Research Workshop, pp. 63-78, 2009.

[39] M. Boehner, "LOGEX - an automatic logic extractor form transistor to gate level for CMOS technology," DAC, pp. 517-522, 1988.

[40] S. Sirowy, G. Stitt, and F. Vahid, "C is for circuits: capturing FPGA circuits as sequential code for portability," ACM/SIGDA Symposium on Field Programmable Gate Arrays, pp. 117-126, 2008.

[41] N. G. Bourbakis, A. Mogzadeh, and S. A. Mertoguno, "Knowledge-based expert system for automatic visual VLSI reverse-engineering: VLSI layout version," IEEE Transactions on Systems, Man and Cybernetics, vol. 32, no. 3, pp. 428-436, 2002.

[42] R. Torrance and D. James, "Reverse engineering in the semiconductor industry," Custom Integrated Circuits Conference, pp. 429-436, IEEE, 2007.

[43] R. Torrance and D. James, "The state-of-the-art in IC reverse engineering," CHES, vol. 5747, pp. 363-381, 2009.

[44] D. Saab, V. Nagubadi, F. Kocan, and J. Abraham, "Extraction base verification method for off the shelf integrated circuits," Quality Electronic Design, pp. 396-400, 2009.

[45] R. L. Rivest and R. E. Schapire, "Inference of finite automata using homing sequences," ACM Symposium on Theory of computing, pp. 411-420, 1989.

[46] M. Kearns and L. G. Valiant, "Cryptographic limitations on learning boolean formulae and finite automata," in ACM Symposium on Theory of computing, pp. 433-444, 1989.

[47] D. Angluin, "Learning regular sets from queries and counterexamples," Information and Computation, vol. 75, no. 2, pp. 87-106, 1987.

[48] R. Nijssen and J. Jess, "Two-dimensional datapath regularity extraction," ACM/SIGDA International Symposium on Physical Design, pp. 42-47, 1997.

[49] D. S. Rao and F. J. Kurdahi, "On clustering for maximal regularity extraction," IEEE Transactions on CAD, vol. 12, no. 8, pp. 1198-1208, 1993.

[50] D. K. Schroder, "Negative bias temperature instability: What do we understand?" Microelectronics Reliability, vol. 47, no. 6, pp. 841-852, 2007.

[51] J. X. Zheng and M. Potkonjak, “Securing netlist-level FFPGA design through exploiting process variation and degradation,” FPGA, pp. 129-138, Feb. 2012.

[52] S. Borkar, T. Karnik, S. Narenda, J. Tschanz, A. Keshavarzi, and V. De, “Parameter variations and impact on circuits and microarchitecture,” DAC, pp. 338-342, 2003.

[53] L. Cheng, J. Xiong, L. He, and M. Hutton, “FPGA performance optimization via chipwise placement considering process variations,” FPL, pp. 1 -6, Aug. 2006.

[54] A. Kumar and M. Anis, “FPGA design for timing yield under process variations,” IEEE Transactions on VLSI Systems, vol. 18, pp. 423-435, March 2010.

[55] B. C. Paul, K. Kang, H. Kufluglu, M. A. Alam, and K. Roy, “Impact of NBTI on the temporal performance degradation of digital circuits,” IEEE Electron Device Letters, vol. 26, pp. 560-562, Aug. 2005.

[56] F. Dabiri and M. Potkonjak, “Hardware aging-based software metering,” DATE, , pp. 460-465, 2009.

[57] S. Meguerdichian and M. Potkonjak, “Device aging-based physically unclonable functions,” DAC, pp. 288-289, 2011.

[58] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar, “Trojan detection using IC fingerprinting,” IEEE Security, pp. 296-310, May 2007.

[59] M. Tehranipoor and F. Koushanfar, “A survey of hardware trojan taxonomy and detection,” IEEE Design & Test of Computers, vol. 27, pp. 10-25, Jan. 2010.

[60] S. Wei, S. Meguerdichian, and M. Potkonjak, “Malicious circuitry detection using thermal conditioning,” IEEE Transactions on Information Forensics and Security, vol. 6, pp. 1136-1145, Sept. 2011.

[61] S. Wei and M. Potkonjak, “Scalable consistency-based hardware Trojan detection and diagnosis,” NSS, pp. 176-183, Sept. 2011.

[62] M. Potkonjak, A. Nahapetian, M. Nelson, and T. Massey, “Hardware trojan horse detection using gate-level characterization,” DAC, pp. 688-693, 2009.

[63] S. Wei, K. Li, F. Koushanfar, and M. Potkonjak, “Hardware trojan horse benchmark via optimal creation and placement of malicious circuitry,” DAC, pp. 90-95, 2012.

[64] S. Wei and M. Potkonjak, “Wireless security techniques for coordinated manufacturing and on-line hardware Trojan detection,” WISEC, pp. 161-172, 2012.

[65] S. Wei and M. Potkonjak, “Scalable hardware Trojan diagnosis,” IEEE Transactions on VLSI Systems, vol. 20, no. 6, pp. 1049-1057, 2012.

[66] S. R. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou, “Designing and implementing malicious hardware,” LEET, pp. 1-8, Apr. 2008.

[67] L. Lin, M. Kasper, T. Guneysu, C. Paar, and W. Burleson, “Trojan side-channels: Lightweight hardware Trojans through side-channel engineering,” CHES, vol. 5747, pp. 382-395, 2009.

[68] J.-F. Gallais et al., “Hardware Trojans for inducing or amplifying side-channel leakage of cryptographic software,” Trusted Systems, vol. 6802, pp. 253-270, 2011.

[69] J. Gaisler, E. Catovic, M. Isomaki, K. Glembo, and S. Habinc, “GRLIB IP core user's manual,” Gaisler Research, 2007.

[70] R. Weicker, “An overview of common benchmarks,” Computer, vol. 23, pp. 65-75, 1990.

[71] “Coremark, an EEMBC benchmark,” 2012.

[72] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, “MiBench: A free, commercially representative embedded benchmark suite,” IISWC, pp. 3-14, 2001.

[73] P. Deutsch and J.-L. Gailly, “ZLIB Compressed Data Format Specification version 3.3.” RFC 1950 (Informational), 1996.

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

ALU	Arithmetic logic unit
ALUOP	Control signals for an ALU
ASIC	Application specific integrated circuit
BHT	Benign hardware Trojan
CMOS	Complementary metal-oxide-semiconductor
CRC-32	Circular redundancy check
DFF	D-type flip-flop
DRM	Digital rights management
DSP	Digital signal processing
EEMBC	Embedded microprocessor benchmark consortium
EEPROM	Electrically erasable programmable read-only memory
FFT	Fast Fourier transformation
FPGA	Field programmable gate arrays
FPU	Floating point unit
FSM	Finite state machine
GPL	GNU public license
GPR	General purpose register
HTH	Hardware Trojan horse
IC	Integrated circuit
IP	Intellectual property
JTAG	Joint test action group
LEON3	32-bit CPU microprocessor core based on SPARC V8 architecture
LUT	Lookup table
ML505	A Xilinx design board
NBTI	Negative bias temperature instability
NMOS	N-type metal-oxide-semiconductor
NRE	Non-recurring engineering
OpenRISC	An open source project developing general purpose RISC CPU architectures.
OR1200	Five state

PMOS	P-type metal-oxide-semiconductor
PUF	Physical unclonable function
RE	Recurring engineering
RS-232	Standard for serial communication
SE	ShuffleEn
SPARC V8	RISC instruction set architecture
SR	Set/Reset
UART	Universal asynchronous receiver/transmitter
v5lx50	A Xilinx Virtex-5 FPGA
VHDL	VHSIC hardware description language