A Flexible Design Flow for Software IP Binding in FPGA

Michael A. Gora, Abhranil Maiti, and Patrick Schaumont, *Senior Member, IEEE*

Invited Paper

***Abstract*—Software intellectual property (SWIP) is a critical component of increasingly complex field programmable gate arrays (FPGA)-based system-on-chip (SOC) designs. As a result, developers want to ensure that their Software Intellectual Property (SWIP) is protected from being exposed to or tampered with by unauthorized parties. By restricting the execution of SWIP to a single trusted FPGA platform, SWIP binding addresses developers' concerns about maintaining control of their intellectual property and the market position it affords.**

This work proposes a novel design flow for SWIP binding on a commodity FPGA platform lacking specialized hardcore security facilities. We accomplish this by leveraging the qualities of a Physical Unclonable Function (PUF) and a tight integration of hardware and software security features. A prototype implementation demonstrates our design flow's ability to successfully protect software by encryption using a 128 bit FPGA-unique key extracted from a PUF. Based on this proof of concept, a solution to perform secure remote software updates, a common challenge in embedded systems, is proposed to showcase the practicality and flexibility of the design flow.

***Index Terms*—Design flow, firmware, field programmable gate arrays (FPGA), intellectual property, physical unclonable function, secure embedded systems, security, software binding.**

I. Introduction

EMBEDDED systems and system-on-chip (SOC) designs based on field programmable gate arrays (FPGA) are becoming increasingly complex in nature, requiring sophisticated software development. As a result, developers need to protect their Software Intellectual Property (SWIP) from counterfeiting, reverse engineering, and tampering. This is highlighted by the estimated 5%–10% of high technology products [3], [26], [29] on the market that are counterfeit, a trend which continues to increase [27], [28]. However, the threat faced by developers goes far beyond an initial reduction in sales. Inferior counterfeit products may perform poorly or malfunction, sometimes catastrophically, resulting in increased service and recall costs [30], [31]. Worse still, are instances of legitimate products containing malicious software [32], [33] introduced through tampering or reverse engineering. Scenarios such as these can rapidly erode customer confidence in a developer resulting in untold damage to their market position.

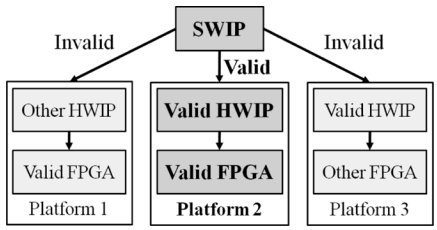

Fig. 1. SWIP binding to a platform comprised of HWIP and an FPGA.

In FPGA design, hardware intellectual property (HWIP) is defined as the soft-core (synthesized from HDL) hardware modules stored in the FPGA configuration bitstream (herein referred to as bitstream). The FPGA design can be protected by means of bitstream encryption, an option offered by several FPGA manufacturers. However, bitstream encryption is not a comprehensive solution. When the FPGA configuration contains programmable components (such as a soft-core processor), the SWIP implemented on top of that soft-core processor requires separate protection.

One solution to address this issue is to encrypt the SWIP and restrict its execution to a specific FPGA. We use the term "Software Intellectual Property Binding" to express this. Two components are considered in such a solution: the SWIP and the hardware platform. SWIP binding ensures that the SWIP will function only when it's deployed on an authentic platform, which includes an authentic (designated) FPGA and a valid (designated) HWIP. Fig. 1 illustrates that the SWIP only functions correctly when an authentic HWIP and a valid FPGA are present, such as with platform 2. Platform 3 fails because the FPGA device is not authentic, while platform 1 fails because the HWIP is not valid. The identity of a design is thus formed by the combination of a FPGA and a HWIP. In this work, we propose an end-to-end design flow for binding a SWIP to a design based on a commodity FPGA.

SWIP binding can be achieved using costly mechanisms such as secure ROM or flash memory to store FPGA specific cryptographic keys. However, this is not only expensive but rules out many commodity and legacy systems as well as being vulnerable to attack [8], [9]. In this work, we instead utilize the ability of a PUF to generate a FPGA-unique secret volatile key to achieve SWIP binding. As a PUF can be deployed securely

Manuscript received November 30, 2009; revised March 31, 2010; accepted August 02, 2010. This work was supported in part by the National Science Foundation under Grant N0 0644070 and in part by the Institute for Critical Technology and Applied Science (ICTAS). Paper no. TII-09-11-0347.

The authors are with Secure Embedded Systems, Bradley Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University, Blacksburg, VA 24060 USA (e-mail: gora@vt.edu; abhranil@vt.edu; schaum@vt.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TII.2010.2068303

in the fabric of a FPGA, our solution avoids the need for costly specialized hardware, and as a result, is suitable for commodity FPGA.

Our main contributions in this paper are as follows.

- To propose a complete, end-to-end design flow to bind a SWIP to a FPGA utilizing a PUF.
- To develop necessary tools for the proposed design flow such as the Intellectual Property Encryptor (IPE) and an obfuscated ROM.
- To demonstrate a complete prototype implementation using our design flow to prove the validity of our idea. To our knowledge, this is the first work to demonstrate such an implementation.
- To propose a secure remote software update scheme for embedded systems utilizing our design flow as a basis to showcase its practicality and flexibility.

The remainder of this paper is organized as follows. Section II provides an overview of related work. Our proposed design flow is presented in Section III. In Section IV, the supporting hardware architecture is described. We present a security analysis of our design flow in Section V. Our design metrics and results are evaluated in Section VI. Section VII presents our proposed scheme for secure remote software updates based on our design flow. Finally, we conclude this paper in Section VIII.

II. Related Work

Software intellectual property binding seeks to limit the execution of SWIP to a particular authorized device or system [15]. Similar to other forms of intellectual property protection, the main goal of this restriction is to prevent the unauthorized duplication or reverse engineering of software. Numerous protection schemes have been proposed to address this issue including watermarking, tamper-resistance, and obfuscation [16]. Tamper resistance allows a program to validate its own integrity and to cease operation if it has been modified [17]–[19]. Watermarking incorporates a developer signature into a program to detect intellectual property theft and reuse [20], [21]. Obfuscation transforms a program in such a way that it is hard to reconstruct the source or assembly code from a static program image [22], [23]. However, when these approaches are deployed purely in a software context, they are vulnerable to virtualization techniques [15] and exploitable features in modern processors such as virtual memory support [24]. To address this issue, a protected execution environment is necessary for many of these techniques.

Any system attempting to provide a protected environment for storage and execution of software must operate as a trusted device. This concept of trusted computing is implemented by the Trusted Computing Group's (TCG) Trusted Platform Module (TPM) [2]. A TPM functions by providing a secure hardware medium for storage and generation of keys or certificates, mechanisms for monitoring a processor, and identifying a system. This concept of a secure platform is further expanded by AEGIS architecture proposed by Suh *et al.* [25]. AEGIS provides for a single-chip processor with built-in tamper resistance and detection as well as dedicated cryptographic components. Alternatively, Lee proposes a Secret Protected (SP) architecture [35] which unlike AEGIS or TPM represents a paradigm shift, where data is tied to a user rather than a particular device. The SP architecture is an attractive solution as its operation is primarily software driven by a Trusted Software Module (TSM). However, to ensure the TSM's integrity a processor must support a Concealed Execution Mode (CEM) that isolates TSM functionality and that contains volatile cryptographic keys. The hardware supported CEM allows the TSM to perform a trusted bootstrap, as proposed in [2], [36], [37], and verify the TSM binary and system state. Such a hardware assisted bootstrap mechanism is considered critical to establish a trusted execution platform that is robust against a variety of attacks.

Lie *et al.* [48] provides an alternative trusted execution platform through the concept of execute only memory (XOM). The XOM architecture ensures certain private portions of memory once written are only executable and can't be modified or read. However, like AEGIS and SP, XOM requires a specialized hardware implementation and a shift from traditional architectures.

This work relies on a PUF as the root of trust. A PUF is a platform-unique function which, when supplied with an input challenge, produces an output response. The response is determined by the behavior of a complex, unclonable physical system, such as the delay variation of logic and interconnects in an FPGA due to manufacturing process variations. It can be used to authenticate chips and generate a volatile secret key required for cryptographic operation without the need of an expensive nonvolatile memory [1]. It is also useful in SWIP protection [3] as well as in securing private information in many applications.

Several different types of PUF have been proposed so far. A ring-oscillator (RO)-based PUF [1] is of particular note among them because of its easy implementation on the FPGA. The complex nature of an FPGA [14] provides a platform to deploy traditional forms of intellectual property protection such as tamper-resistance and obfuscation. By coupling a PUF with these techniques, we can provide an efficient platform for binding SWIP to an FPGA.

In the work of Guajardo *et al.* [12], a SRAM-based PUF protection mechanism is proposed for securing HWIP modules. In contrast, our work focuses on protecting SWIP and provides a demonstration system that explores generation and encryption of the protected SWIP. In addition, we also provide a detailed mechanism that addresses how to perform the parsing, decryption, and loading of encrypted SWIP sections. Guajardo assumes the existence of such a hardware mechanism and does not go into detail about the nature of such a component.

An updated Aegis architecture has been proposed in [13] for secure software execution using PUF. That work proposes the use of PUF for runtime memory-integrity through the use of hash trees. Our approach addresses configuration of SWIPs, and shows how to authenticate them onto an FPGA fabric.

III. SWIP Binding Design Flow

We propose a generic design methodology that aims to bind any SWIP to an FPGA platform irrespective of the class or vendor of the FPGA.

A. Overview

Our design flow is introduced in the context of the different phases in FPGA-based system design. In Sections IV–VIII, we

define the system components, the parties involved in the design flow, and the model of trust that governs their interactions.

1) System Components: Typically, the end-user only perceives a finished product, while the system developer has three components to manage.

- FPGA—The physical silicon that the HWIP and SWIP are deployed on. It might contain a hard core processor or other specialized hardware such as multipliers.
- HWIP—The soft core processor and other hardware components, including the PUF, which are configured into the FPGA fabric. This is typically the level at which HWIP protection/binding schemes are implemented, for example with bitstream encryption [11].
- SWIP—The application that executes on a soft core or hard core processor in the FPGA (herein referred to as binary). This software is often stored outside of the FPGA bitstream. We target this component for binding to an FPGA.

2) Involved Parties: For the purpose of our design flow, we define the parties involved in system development and use as follows.

- FPGA Manufacturer—The manufacturer of the physical FPGA such as Xilinx or Altera.
- System Developer—The developer of the system on which the SWIP will run. This may include the physical embedded system or just the HWIP contained on the FPGA. The system developer handles the creation and application of the PUF. In certain instances the system developer and FPGA manufacturer can be the same party.
- SWIP Developer—The developer of the end-user applications (SWIP) which operate on the FPGA-based embedded system and it's HWIP.
- End User—The system customer who will utilize the system as either a standalone product or as part of a larger integrated system.

3) Trust Model: The primary concern of the FPGA manufacturer is the sale of FPGA hardware. As a result, it is in the manufacturer's interest to provide a secure platform that is more appealing to system or SWIP developers. Likewise the goal of the system developer is to produce a robust full featured platform to attract the end user. To this end, it is in the system developer's interest to provide a secure, tamper resistant, platform for SWIP developers. The goal of the SWIP developer is to maximize revenue from the sale of its SWIP. Thus, the system that can best provide security against counterfeiting or reverse engineering will attract the best end user applications and generate better sales. Both the FPGA manufacturer and the system developer benefit from increased sales and have an incentive to trust each other. Beyond financial incentives the interleaved nature of an FPGA-based embedded system requires that the FPGA, HWIP, and SWIP are considered trusted. To this end we assume that the bitstream is obfuscated, the SWIP is well written (not prone to vulnerabilities), that the FPGA is tamper resistant, and that decrypted SWIP is only stored in internal FPGA memory (not external RAM). Fig. 2 illustrates these overlapping trust boundaries by the dashed regions.

Often an end-user is only interested in obtaining the best bargain which can include counterfeit systems or pirated SWIP. Worse still is the case when the end-user is malicious, actively seeking to compromise the design for the purposes of reverse engineering or counterfeiting. We base our design flow on this adversarial model which is represented in the exclusion of the end user in Fig. 2. This includes any software that is not directly bound or verified by the design flow (i.e., belong to the trusted parties) and may possibly be malicious.

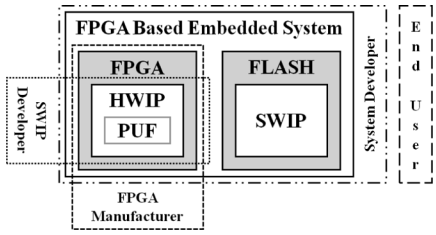

Fig. 2. Trust boundary and relationships between system components and development parties. Dashed regions which overlap signify trusted relationship between parties.

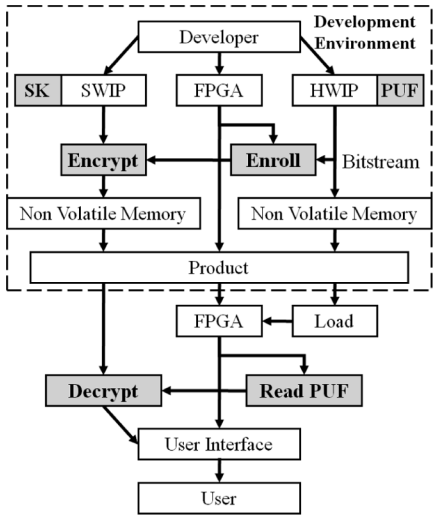

Fig. 3. FPGA system design flow from developer to end-user.

B. Design Overview

Fig. 3 illustrates how we employ our PUF-based SWIP binding methodology in the context of FPGA-based system design. The elements in gray represent the major contributions of our approach and are broadly divided in two parts.

- Before delivery to the end-user, inside the trusted and secure development environment, an FPGA-unique key is extracted from the PUF in a process called enrollment. The SWIP is then bound to the FPGA by encrypting it using the PUF key with the help of a custom encryption tool.
- After delivery to the end-user, when the FPGA boots up, a security kernel (SK) extracts the PUF-based key from the FPGA to decrypt the encrypted SWIP for execution.

The FPGA, its configuration bitstream (including the PUF and HWIP) and the protected binary (including the encrypted SWIP and the SK) constitute the final product to the end-user. It is critical to prevent an attempt to modify the components of the delivered product for the purpose of extracting the PUF key. To solve this problem, we implemented an integrity mechanism which is discussed in detail in Section III-E.

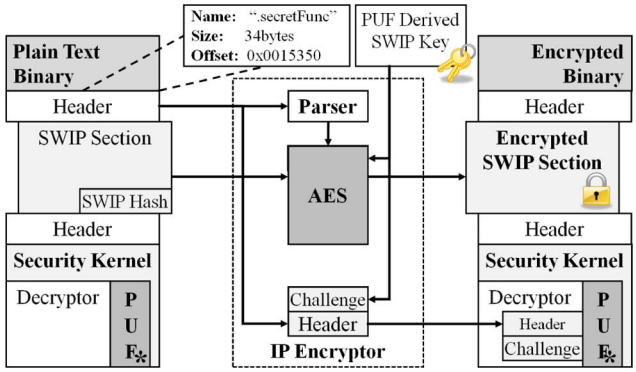

Fig. 4. SWIP binding encryption process as performed by the PC-based IPE. PUF software interface is denoted by the asterix (*).

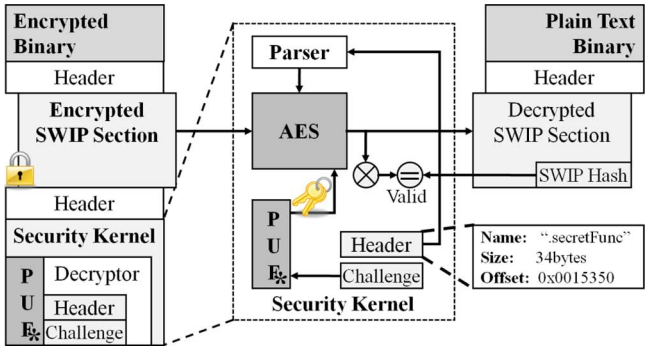

Fig. 5. SWIP binding decryption process as performed by the SK deployed on the embedded system. PUF software interface is noted by the asterix (*).

C. SWIP Binding

Ideally, all software would be encrypted and verified (hashed) to prevent any analysis or tampering by an adversary. However, a system cannot boot from a completely encrypted executable. Therefore, there must be a mechanism which will first extract the PUF-based key and decrypt the SWIP before execution can begin. Though specialized hardware may be employed to perform this task, such a system would not only be resource intensive in terms of FPGA area but would also hinder system flexibility. Instead, a software-based Security Kernel (SK) is introduced that remains unencrypted (plain text). The result is a system that contains two major types of software. One is the SWIP and must be encrypted and the other is the SK that remains in plain text.

Fig. 4 shows an unprotected binary containing SWIP and our SK in the .elf binary format used by many compilers. The mixed nature of the software in our system requires that encryption is only performed on certain sections. To facilitate this, we created a standalone Intellectual Property Encryption (IPE) PC-based utility that can extract and parse the section headers from the binary. These headers contain information about the name, location, and size of the software in that particular section. The C attribute functionality is used to specify the name for sections related to the SK. All other sections, regardless of their content, are considered SWIP sections. Based on the header information and our naming convention, the IPE retrieves the SWIP section from the binary file. A simple hash of the section is performed and stored within the section before encryption is performed (on the section and its hash) through 128-bit AES in counter mode [4] using a key derived from the PUF. The IPE performs the encryption of the entire software binary inside the trusted development environment and is itself not deployed to the hardware system.

Additionally, the IPE inserts plain text information into the SK including the PUF challenge and the header information. In the end, a protected binary is produced containing both the encrypted SWIP and the plain text SK section.

D. Boot Procedure

At boot time, when the protected binary is downloaded to an FPGA, the SK performs the operations of the IPE in reverse, as shown in Fig. 5. Utilizing the challenge stored in it by the IPE, the SK retrieves the PUF key. Next the security kernel parses the encrypted binary using the header information included by the IPE. Once a section of SWIP is located, it is decrypted using 128-bit AES with the key retrieved from the PUF and the results stored in the FPGA's internal memory. To ensure an error free decryption, the SK performs a simple hash of the decrypted SWIP section. This value is compared to a hash performed by the IPE and included at the end of each SWIP section. If the hash values differ this could indicate a compromised section or a failed decryption. After decryption is completed and validated, the SK turns execution over to the SWIP. If all sections of SWIP are decrypted then the memory occupied by the SK can be reallocated for other uses.

E. Trusted Boot

A key concern during the boot procedure is maintaining the integrity of the SK and ensuring that only its validated software executes. This is necessary to ensure that the system boots in a trusted state and requires that any tampering to the boot or interrupt vectors as well as to the SK software be detected. However, the drawback of our software-based scheme is that the SK can't be trusted because a plain text binary can be read and modified. As the security kernel is not considered as SWIP, its confidentiality is not of concern. The greater issue is that a compromised security kernel could be utilized to retrieve the PUF key, and the decrypted SWIP. We address this issue with the inclusion of an Integrity Kernel (IK).

1) Integrity Kernel: Validation of the security kernel is the primary function of the IK. A hashing algorithm is commonly used to establish the validity of software by comparing the results of the hash with a reference value. We boot our system in the IK which runs a hash on the security kernel and validates it against a reference result. By verifying that it has not been tampered with, the execution can pass to the security kernel, and then it can begin decryption of the SWIP.

It is imperative that the IK and the boot procedure are secure against attacks. If an attacker can bypass either the IK or SK they could potentially execute untrusted software and extract the key from the PUF. To achieve a trusted boot (Fig. 6), we introduce an obfuscated ROM to provide a tamper resistant storage mechanism for the IK and all values needed to control the boot process. This prevents an attacker from subverting or surpassing the IK which in turn verifies that the SK has not been modified.

2) Obfuscated ROM: An FPGA bitstream is believed to have an inherent layer of obfuscation. Though LUT configuration

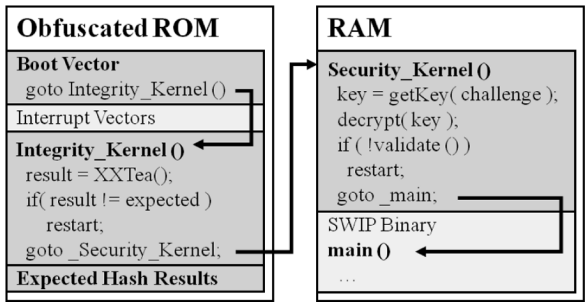

Fig. 6. Trusted boot procedure through IK verification of the SK.

and BRAM contents can be accessed relatively easily, a complete reverse engineering of the bitstream into a net-list has not been reported so far [14]. Based on this assumption, an obfuscated ROM is implemented using multilevel logic in the FPGA avoiding direct storage in LUTs or BRAMs. This ROM is used to securely store the IK binary. Even though the required logic circuits are implemented using LUTs in an FPGA, the contents of the ROM cannot be extracted just by reading the contents of the LUTs. This is because the ROM circuit is formed by a combination of several LUTs which is spread over the FPGA, and interleaved with other circuit components.

The IK binary includes the IK software, the boot vectors, and the interrupt vectors. The vectors are included into this binary to help protect the software execution flow. These values are hard coded as the content of the obfuscated ROM. Since synthesizing the obfuscated ROM is trivial, modifying the design of the integrity kernel is not difficult. This helps to maintain the flexibility of our design flow. However, it is important to maintain a low footprint, while implementing the ROM.

3) Hashing Algorithm: Employing an obfuscated ROM is costly in terms of area and is primarily why it is not used to deploy the SK. As a result, selection of a compact hashing algorithm with small memory footprint is essential.

Traditional hashing algorithms such as SHA-1 are large due to the size of their internal state. An alternative solution is the use of a cipher-based Davies–Mayer hash which allows us to leverage the compact nature of certain ciphers. XXTea for example can be deployed in under 380 bytes in such a configuration [5], [6]. Combined with initialization data and the expected results of the hash, we are able to implement it in a 512 byte block of obfuscated ROM.

F. Design Summary

All the software and hardware components, required for our design flow, are illustrated in Fig. 7 from a developer's perspective with shaded figures as main components of our design flow.

Our proposed system is able to achieve a high flexibility for several reasons. First, we do not rely on any specific hard core facilities or capabilities in an FPGA. Our design only requires the ability to deploy a soft core processor. Only the interface with the hardware PUF would be system specific. Second, by maintaining the majority of our functionality in software, we ensure rapid substitution of components such as the PUF, error correction, and various cryptographic primitives to meet the specific needs of the developer. Finally, by developing our design using standard C libraries we ensure compatibility across a wide array of soft and hard core processors.

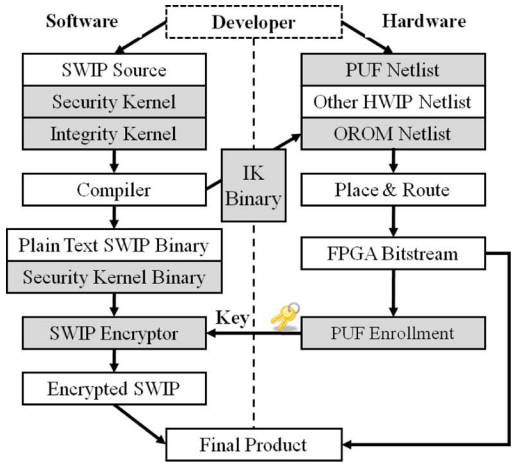

Fig. 7. Developers design flow, components specific to SWIP binding are shaded gray.

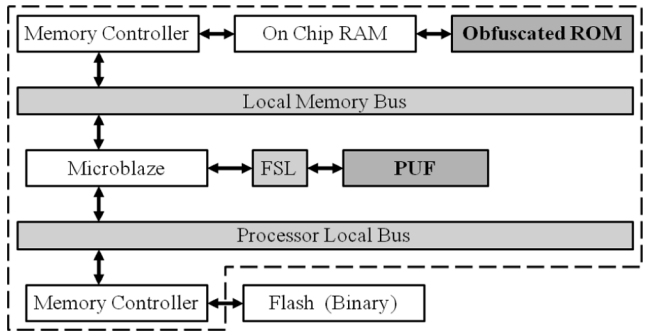

Fig. 8. SWIP binding prototype system architecture on a Xilinx Spartan XC3500E FPGA.

IV. Demonstration System

This section presents a basic proof of concept demonstration system for our design flow. We assume the binding of a single SWIP binary to an embedded system during a trusted production process. The underlying hardware architecture has been selected with this scenario in mind.

A. FPGA Hardware Architecture

Fig. 8 shows the different components of the hardware architecture used for our prototype implementation on a Xilinx Spartan XC3S500E FPGA. A Microblaze soft core processor integrates several co-processors attached through a 32-bit processor local bus. The PUF is attached as a co-processor to the Microblaze using a dedicated fast simplex link (FSL). The block rams (BRAM) are not used as on-chip memory. Instead, the obfuscated ROM is used to achieve secure boot as discussed in Section III-E. The dotted box in Fig. 7 indicates the boundary of the FPGA. Anything outside it is an off-chip component, and is nontrusted.

B. Physical Unclonable Function (PUF)

For our prototype implementation, we used an RO-based PUF that has been proposed in [1] using several identical ROs. This PUF exploits random but static manufacturing process variations in RO frequencies. The PUF output is created by pair-wise comparison of the RO frequencies. These comparisons can be

represented as a challenge/response function, where the chosen ring oscillator pair is the challenge, and the comparison result is the response. An RO PUF has been selected due to its suitability for an FPGA implementation.

A single RO circuit is created as a hard macro and instantiated several times to build the PUF as proposed in [1]. Enrollment of the PUF, the process of extracting the challenge/response pairs for the first time, is performed by simple C-program during the trusted development phase. The encryption keys, required for SWIP encryption, are derived from the PUF enrollment.

C. Error Correction of PUF

PUF outputs are noisy by nature; a finite number (though low) of PUF output bits vary over time. However, for cryptographic operation, a stable key is necessary. Generating noise free keys from PUF is an active area of research, and several error correction schemes have been proposed such as [45]. Most of them require a complex implementation in software and/or hardware. The main objective in this work is to show the use of a PUF in the SWIP protection mechanism, and therefore, we have implemented a simple but effective error correction mechanism. By only selecting the RO pairs that have a relatively high difference in frequency we can extract a stable key and ignore error prone RO pairs. The drawback of this scheme is that we are not able to fully utilize all the ring oscillators for key extraction. However, other error correction techniques can be employed in a production scenario.

V. Security Analysis

At first, we assume that the developer's environment will protect both HWIP and SWIP sources, including the PUF design (as defined in Fig. 2). Based on this assumption, we make an effort to achieve SWIP binding in the untrustworthy user environment.

The goal of the attacker is to reveal the secret key in order to decrypt the SWIP. In our method, the PUF-based key remains internal to the FPGA and never gets exposed, so the attacker has to try to modify either the software or the hardware platform consisting of the FPGA device and the configured HWIP in order to determine key. Based on this assumption, we discuss a few relevant hardware and software security issues.

A. Hardware Analysis

1) Physical Attacks: If a physical attack is mounted on the FPGA device such as by laser cutting or removing chip layers, it is believed in current literature that the complex and sensitive delay behavior of the PUF changes and the key is destroyed.

2) Bitstream Attacks: As our system does not utilize a specialized bitstream protection scheme, the bitstream containing the PUF is visible to an attacker. However, due to the complex nature of an FPGA bitstream; it possesses an inherent layer of obfuscation. We are well aware that 'security by obscurity' is not a solid strategy. We can however point out the relative difficulty of the problem of bitstream reverse engineering by considering related work. Most advanced current work can only produce a flat netlist of design elements from a bitstream [50]. Reconstruction of the design, and identifying the exact location of the PUF, requires the attacker to reverse engineer the netlist and restore at least one level of hierarchy. Additional work in bitstream manipulation for partial reconfiguration shows that such manipulation requires strict constraints and detailed design knowledge [51]. Hence, also "patching" of the bitstream to leak the PUF key is not a straightforward activity. We also make the practical observation that perfect security does not exist. Based on a physical instance of the chip, a single device can be completely reverse engineered up to the schematic level [52]. However, such an operation costs a significant amount of money and effort. We thus believe that at the moment of writing, bitstream reverse engineering is still sufficiently hard and uncommon so that a PUF itself can serve as a root of trust.

B. Software Analysis

1) Software Integrity: As the SWIP is hashed after decryption to verify its integrity it is very difficult to modify the protected binary in any useful way. On the other hand, since the SK is in plain text, it could be modified. However, the addition of the IK using the obfuscated ROM prevents the execution of the SK if it has been altered.

2) Execution of Malicious Code: An attacker could attempt to subvert a deployed system in such a way that they could execute malicious code to extract the key. We prevent such an attack by the application of obfuscated ROM inside the bitstream which contains the boot/interrupt vectors for the processor as well as the IK. The IK stored in the obfuscated ROM verifies the integrity of the SK before it is allowed to execute. Likewise, the SK decrypts and verifies the integrity of SWIP before it allows it to begin execution. The only way an attacker can then subvert this protection is by successfully reverse engineering and altering the bitstream in a meaningful way, attacking the implementation of the SWIP (buffer overflow, etc.), or directly attacking physical weaknesses of system hardware. However, we note in the discussion of our trust model and security analysis that we assume that the bitstream is obfuscated, the SWIP is well written (not prone to vulnerabilities), and that the FPGA is tamper resistant.

3) Other Attacks: Finally, there is the concern for traditional methods of attacking software at runtime such as buffer overflow or exploitation of inherent weaknesses in the software. In general, SWIP binding can do little to avoid such issues. Rather, the SWIP that is being protected must be validated as being well written to avoid such problems. After the initial secure boot, no guarantees can be made to the state or integrity of the SWIP at runtime.

C. System Design Tradeoff

As our system is designed to provide a flexible framework for SWIP binding we only specify the general functionality of certain components such as the PUF error correction. It is important that the implementations selected for these system components provide adequate security. Specifically, error correction schemes should not leak information about the key and maintain a good source of entropy. Similarly, the output of a PUF should be unique across devices and show a high level of stability [1].

TABLE I
PUF METRICS

# of RO's	Uniqueness	Reliability	Enrollment Time	Runtime Extraction
256	44%	96.7%	90 Seconds	4 Seconds

VI. RESULTS

In this section, we present our prototype results, including an assessment of the system demonstrator, the characterization of the PUF, and resource usage.

A. Functionality Testing

We have tested our prototype design on five Xilinx Spartan XC3S500E FPGAs. PUF enrollment was done on all of the FPGAs to extract their respective 128-bit keys and a single C-code binary was encrypted using each of these keys to produce five encrypted binaries. To validate the SWIP binding functionality, these five binaries were executed individually on each of the five sample FPGA chips. Each of the FPGAs could successfully execute exactly one binary with no two of them being able to execute the same binary.

B. PUF Characterization

We measure two parameters namely uniqueness and reliability to characterize the PUF. We define uniqueness as an estimate of how clearly a PUF can distinguish an FPGA from another. In other words, it estimates the difference between two keys generated by the PUF on two different FPGAs. Reliability expresses the stability of a specific response that is produced by a PUF from an FPGA for a given challenge. A stable PUF should reproduce the same response with minimum rate of error for a particular challenge if the challenge is applied to the PUF multiple times. Table I shows the metrics of the PUF that we implemented.

The PUF performs with a high reliability, while giving a moderately high value of uniqueness. The reliability figure of 96.7% is calculated for all 255 pairs of ring oscillators during the PUF enrollment although a 100% reliable key is generated using the most stable 128 pairs as described in the PUF error correction (4.3).

Enrollment requires 90 s to generate a stable key on average. However, it is a onetime operation. On the other hand the time required for runtime key extraction is 4 s. We provide these figures to verify the functionality of our proof of concept system and refer to Maiti *et al.* [45] for a more detailed evaluation of the reliability and stability RO-based PUFs.

C. Hardware Utilization

Table II shows the overall FPGA slice count used for the whole design is 3678, although the components specific to our design flow (i.e., the PUF and the ROM) only need 745 slices.

D. Security Software Overhead

As evident from the Table III, the software memory overhead required for our particular implementation can be considered sizeable for the overall internal memory of a Spartan XC3S500E FPGA. However, this implementation is a proof of concept. Our design flow is flexible to allow different hash and decryption primitives to be utilized to achieve smaller code size, higher security or faster execution time.

TABLE II
RESOURCE UTILIZATION XILINX SPARTAN XC3S500E

Component	LUTs	Slices
Microblaze	1397	698
PUF	931	279
Obfuscated ROM	559	279
DDR Ram Controller	1304	1024
Total	4563	3678

TABLE III
SK AND SK MEMORY REQUIREMENTS

Design Flow Element	Component	Size (Bytes)
Security Kernel	AES	10560
	Total	14720
Integrity Kernel	Boot/Interrupt Vectors	80
	XXTEA	376
	Expected Hash Results	8
	All Components	464

VII. CASE STUDY

Through our demonstration system and its analysis we have shown that not only is our design flow functional but also practical for deployment on commodity FPGA. However, this demonstration only addresses the very basic case of SWIP binding. To illustrate a key principle in our design methodology, flexibility, we provide a theoretical case study of its application to solve a common problem in FPGA-based embedded systems, software update.

A. Software Update

Embedded system development does not halt when a system enters production. End-user feedback can expose bugs or suggest possible improvements. As a result developers will often support a system with continued software updates to either improve the end-user experience or provide additional income in the case of feature updates. In complex, safety critical, embedded system rich environments, such as automobiles it is imperative that these updates be performed expediently and securely [39], [40]. However, it is impractical to have an end-user return a system to a developer to perform such updates. A more pragmatic approach is performing the updates over a network connection.

A typical remote software update protocol can be broken up into three phases: initiation, authentication, and distribution. First, a remote system requests a software update from a secure update sever. Then, the server generates a challenge for which the remote server must be able to generate an appropriate response for authentication. Next, the SWIP is encrypted and packaged in such a way to ensure integrity, authenticity, confidentiality, and freshness [46]. This must be performed in such a way to minimize storage, computational overhead, and

transmission overhead [47]. Finally, the system must be able to perform such updates multiple times without compromising the security of the PUF-based key storage. Nilsson *et al.* [38] proposes a traditional public key-based approach to providing secure remote updates. In that work, they identify the following security properties that such a system must possess.

- Integrity of software must be maintained to ensure that it has not been modified or corrupted.
- Authenticity of the software must be confirmed as coming from the trusted system developer.
- Software must remain confidential as it can contain valuable intellectual property (SWIP).
- Only the freshest updates must be allowed to prevent an attacker from reusing old updates [46].

We have already shown that our design flow can ensure integrity, authenticity, and confidentiality for SWIP. In addition, we can expand upon Nilsson's approach by extending these principles beyond the transmission of the SWIP to its local non-volatile storage. We address the selection of appropriate protocols and components to securely map our design flow to these properties in Section VIII.

B. SWIP Protection Authentication Protocol

Our demonstration system accomplishes authentication locally during the production process through enrollment. However, after this point, we provide no further mechanism to establish the identity of the system. This is not required due to the assumption that we are only concerned with protecting a single instance of the SWIP once deployed to an FPGA-based embedded system. Several PUF-based authentication protocols have been proposed based on symmetric [3] and public key [42] systems. Of particular interest is a simplified IP protection authentication protocol proposed by Guajardo *et al.* [3]. However, this protocol is concerned with only HWIP modules and doesn't take into account the implicit relationship between the FPGA manufacturer and the IP (HWIP/SWIP) developer. Let us take into consideration the scenario of a malicious hardware manufacturer. In such an instance Guajardo argues it is necessary to prevent the malicious manufacturer from gaining access to the decrypted IP or its key. However, any IP that operates on a given FPGA must at one point exist in a decrypted state on that FPGA. A malicious manufacturer could introduce functionality to extract this IP once it has been decrypted for operation. As a result the trust third party included by Guajardo to act as an intermediary, so that the IP is not exposed to the system developer, is unnecessary. We present a modified version of the protocol in Fig. 9 to reflect these differences.

The protocol is divided into two portions, enrollment and software update. During enrollment the system developer, acting as the trusted party, issues a unique public identifier to the system, IDSYS. Next, an enrollment procedure is performed on the PUF generating a model or list of challenge (C) response (R) pairs (CRP) for authentication which is stored by the system developer.

During software update, the system, in the possession of the end-user, makes a request for a software update by transmitting its system identifier ID_{SYS} and the version of its current software ID_{SW} to the system developer. A request containing

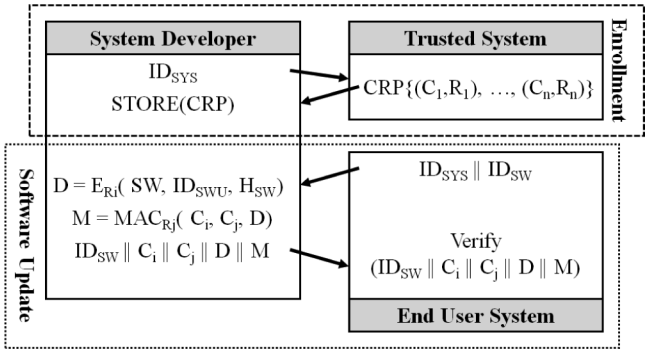

Fig. 9. Modified authentication protocol based on Guajardo's PUF-based symmetric key authentication protocol [3].

the software identifier is forwarded to the SWIP developer. If a newer version of the software SW is available, it is transmitted to the system developer over a secure channel. The system developer chooses a pair of challenges C_i, C_j recorded from the enrollment process and references the corresponding responses R_i, R_j. The response R_i can then be used as a key for encrypting a concatenation of the software update SW, the software update identifier ID_{SWU}, and a hash of the software update H_{SW}.

Integrity and authenticity of the transmitted software update are assured by the generation of a Message Authentication Code (MAC) utilizing a keyed hashing protocol such as HMAC [43]. The MAC is generated for the encrypted software update D and the challenges C_i, C_j using the response R_j as a key. Finally, this is transmitted to the system in possession of the end-user where the software update can be verified as authentic and decrypted. If the software identifier ID_{SW} is a newer version, then the system will update its software.

It should be noted that this protocol only provides for updating the SWIP binary and not the FPGA bitstream. This could allow a new SWIP binary to be deployed on a system with an old compromised FPGA bitstream. Champagne *et al.* [49] offers a solution by directly coupling the FPGA bitstream version with the SWIP binary version. As such the newest SWIP binary will only execute on the most recent FPGA bitstream version.

C. Design Flow Mapping

Support for the authentication protocol can be accomplished with minimal system modifications over our demonstration system, through software only enhancements within the frame of the design flow. These modifications are broken down into two tasks, features and security.

1) Features: Two features need to be included into the system to provide support for the modified authentication protocol.

- The system must include connectivity functionality to support receiving the software update. This can be stored inside of an encrypted software section that is first decrypted by the SK before a software update can be performed.
- The SK must be modified to include support for the verification of the MAC. This is crucial so that the entire can be verified for authenticity and integrity without decryption first.

2) Security: A major concern with the new application of the PUF for remote authentication is that each exchange exposes

a CRP of the PUF. As the PUF only contains a finite (sometimes quite limited) number of CRPs an attacker can attempt a model building attack. Several proposals have been introduced for preventing such attacks but they each rely on disrupting the relationship between input and output of the PUF.

Gassend [44] proposes the application of a random secure hash function on a pre-challenge before it is passed to the PUF and on the response after it is generated. Due to the principles of a secure hash, hashing the prechallenge before it is issued to the PUF makes it difficult for an attacker to specify the challenge the PUF receives. Similarly, passing the response through a hash hides the value of the response from the attacker due to the one way nature of hashes. In this way, such a scheme removes a direct observable relationship between the challenge and response outputs of the PUF. Such functionality can greatly improve a PUF's resilience to attack and is trivial to implement as software interface layer in our SK.

VIII. Conclusion

In this paper, we proposed a flexible design flow that enables binding of software intellectual property to a specific FPGA with the help of a ring oscillator-based physical unclonable function. Validity of the design flow is demonstrated with a prototype implementation. We are also able to show how the properties of a circuit level component such as a PUF can be utilized at the system level using software control. By only utilizing fabric-based hardware structures and highly portable C code we are able to maintain a high degree of flexibility, while still providing adequate security. We demonstrated this flexibility by mapping design flow to address a common problem in FPGA-based embedded systems, remote software update. Coupled with a commodity FPGA our design flow can be used as a stepping stone for developing robust and secure trusted platforms.

References

[1] G. E. Suh and S. Devadas, "Physical unclonable functions for device authentication and secret key generation," in *Proc. 44th ACM/IEEE Design Autom. Conf., DAC'07*, Jun. 4–8, 2007, pp. 9–14.

[2] [Online]. Available: https://www.trustedcomputinggroup.org/home

[3] J. Guajardo, S. S. Kumar, G.-J. Schrijen, and P. Tuyls, "Brand and IP protection with physical unclonable functions," in *Proc. IEEE Int. Symp. Circuits, Syst., ISCAS'08*, May 18–21, 2008, pp. 3186–3189.

[4] N. Ferguson and B. Schneier, *Practical Cryptography*. New York: Wiley.

[5] D. Wheeler and R. Needham, TEA Extensions Cambridge Univ., Cambridge, U.K., Tech. Rep., 1997.

[6] R. Winternitz, "A secure one-way hash function built from DES," in *Proc. IEEE Symp. Informat. Security Privacy*, 1984, pp. 88–90.

[7] Xilinx Embedded Systems Tool Reference Manual.

[8] R. Anderson and M. Kuhn, "Tamper resistance—A cautionary note," in *Proc. 2nd USENIX Workshop Electronic Commerce*, Nov. 1996, pp. 1–11.

[9] S. P. Skorobogatov, "Semi-invasive attacks—A new approach to hardware security analysis," Univ. Cambridge, Cambridge, U.K., Tech. Rep. UCAM-CL-TR-630.

[10] C. Bosch, J. Guajardo, A. Sadeghi, J. Shokrollahi, and P. Tuyls, "Efficient helper data key extractor on FPGAs," in *Proc. Cryptographic Hardware Embedded Syst., CHES'08*, 2008, pp. 181–197.

[11] Using Bitstream Encryption Xilinx. Inc..

[12] J. Guajardo, S. S. Kumar, G.-J. Schrijen, and P. Tuyls, "FPGA intrinsic PUFs and their use for IP protection," in *Proc. Cryptographic Hardware Embedded Syst., CHES'07*, 2007, pp. 63–80.

[13] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas, "Design and implementation of the AEGIS single-chip secure processor using physical random functions," *SIGARCH Comput. Archit. News 33*, pp. 25–36, 2005.

[14] "Volatile FPGA design security—A survey," Saar Drimer, 2008. [Online]. Available: http://www.cl.cam.ac.uk~sd410/papers/fpga_security.pdf

[15] M. J. Atallah, E. D. Bryant, J. T. Korb, and J. R. Rice, "Binding software to specific native hardware in a VM environment: The PUF challenge and opportunity," in *Proc. 1st ACM Workshop Virtual Machine Security* , Oct 27, 2008, pp. 45–48.

[16] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation—Tools for software protection," *IEEE Trans. Softw. Eng.*, vol. 28, no. 8, pp. 735–746, Aug. 2002.

[17] B. Horne, L. Matheson, C. Sheehan, and R. E. Tarjan, "Dynamic self-checking techniques for improved tamper resistance," *Security and Privacy in Digital Rights Management*, vol. 2320, Lecture Notes in Computer Science, pp. 141–159, 2002.

[18] H. Chang and M. Atallah, "Protecting software code by guards," *Security and Privacy in Digital Rights Management*, vol. 2320, Lecture Notes in Computer Science, pp. 160–175, 2002.

[19] H. Jin, G. Myles, and J. Lotspiech, "Towards better software tamper resistance," *Proc. ISC 2005*, vol. 3650, Lecture Notes in Computer Science, pp. 417–430, 2005.

[20] X. Zhang, F. He, and W. Zuo, "Hash function based software watermarking," in *Proc. Advanced Softw. Eng. Appl.*, Dec. 13–15, 2008, pp. 95–98.

[21] J. Palsberg, S. Krishnaswamy, M. Kwon, D. Ma, Q. Shao, and Y. Zhang, "Experience with software watermarking," in *Proc. 16th Annu. Conf. Comput. Security Appl., ACSAC'00*, Dec. 2000, pp. 308–316.

[22] J. Ge, S. Chaudhuri, and A. Tyagi, "Control flow based obfuscation," in *Proc. 5th ACM Workshop on Digital Rights Manage.*, Nov. 7, 2005, pp. 83–92.

[23] W. Michiels and P. Gorissen, "Mechanism for software tamper resistance: An application of white-box cryptography," in *Proc. ACM Workshop Digital Rights Manage.*, Oct. 29, 2007, pp. 82–89.

[24] P. van Oorschot, A. Somayaji, and G. Wurster, "Hardware-assisted circumvention of self-hashing software tamper resistance," *IEEE Trans. Dependable Secure Comput.*, vol. 2, no. 2, pp. 82–92, Apr.–Jun. 2005.

[25] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "AEGIS: Architecture for tamper-evident and tamper-resistant processing," in *Proc. 17th Annu. Int. Conf. Supercomput.*, Jun. 23–26, 2003, pp. 160–171.

[26] J. Stradley and D. Karraker, "The electronic part supply chain and risks of counterfeit parts in defense applications," *IEEE Trans. Components and Packaging Technol.*, vol. 29, no. 3, pp. 703–705, Sep. 2006.

[27] The International Anti_Counterfeiting Directory 2002, Counterfeiting Intelligence Bureau, 2005. [Online]. Available: http://www.iccwbo.org/ccs/cib_bureau/CIBDir.pdf

[28] RF Café—Gray Market (Counterfeit Components), 2006. [Online]. Available: www.rfcafe.com/references/gray_market_links.htm

[29] "Managing the risks of counterfeiting in the information technology industry," White Paper, KPMG Electronics,, 2005. [Online]. Available: http://www.agmaglobal.org/

[30] M. Pecht and S. Tiku, "Bogus: Electronic manufacturing and consumers confront a rising tide of counterfeit electronics," *IEEE Spectrum*, vol. 43, no. 5, pp. 37–46, May 2006.

[31] F. Staake and E. Fleisch, *An Introduction to Counterfeit Markets*. Berlin, Germany: Springer-Verlag, 2008, pp. 3–21.

[32] F. Adelstein, M. Stillerman, and D. Kozen, "Malicious code detection for open firmware," in *Proc. 18th Annu. Comput. Security Appl. Conf.*, 2002, pp. 403–412.

[33] C. E. Irvine and K. Levitt, "Trusted hardware: Can it be trustworthy?," in *Proc. 44th ACM/IEEE Design Autom. Conf., DAC'07*, Jun. 4–8, 2007, pp. 1–4.

[34] B. Jun, "Attack of the clones: Building clone-resistant products," in *Presentation in Proceedings of RSA Conf.*, 2006.

[35] R. B. Lee, P. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang, "Architecture for protecting critical secrets in microprocessors," in *Proc. 32nd Int. Symp. Comput. Architecture, ISCA'05*, Madison, WI, Jun. 4–8, 2005, pp. 2–13.

[36] W. A. Arbaugh, D. J. Farber, and J. M. Smith, "A secure and reliable bootstrap architecture," in *Proc. IEEE Symp. Security, Privacy*, May 4–7, 1997, pp. 65–71.

[37] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and implementation of a TCG-based integrity measurement architecture," in *Proc. 13th Usenix Security Symp.*, Aug. 2004, pp. 16–16.

[38] D. K. Nilsson and U. E. Larson, "Secure firmware updates over the air in intelligent vehicles," in *Proc. IEEE Int. Conf. Commun. Workshops, ICC'08*, May 19–23, 2008, pp. 380–384.

[39] U. E. Larson and D. K. Nilsson, "Securing vehicles against cyber attacks," in *Proc. 4th Annu. Workshop Cyber Security Inform. Intell. Res., CSIIRW'08*, 2008, vol. 288.
[40] U. E. Larson and D. K. Nilsson, "Securing vehicles against cyber attacks," in *Proc. 4th Annu. Workshop Cyber Security Inform. Intell. Res., CSIIRW'08*, 2008, vol. 288, pp. 1–3.
[41] J. Guajardo, S. S. Kumar, G.-J. Schrijen, and P. Tuyls, "Brand and IP protection with physical unclonable functions," in *Proc. IEEE Int. Symp. Circuits Syst., ISCAS'08*, May 18–21, 2008, pp. 3186–3189.
[42] J. Guajardo, S. S. Kumar, G.-J. Schrijen, and P. Tuyls, "Physical unclonable functions and public-key crypto for FPGA IP protection," in *Proc. Int. Conf. Field Programmable Logic Appl., FPL'07*, Aug. 27–29, 2007, pp. 189–195.
[43] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-hashing for message authentication," in *Internet RFC 2104.* 1997.
[44] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas, "Controlled physical random functions," in *Proc. 18th Annu. Comput. Security Appl. Conf.*, 2002, pp. 149–160.
[45] A. Maiti and P. Schaumont, "Improving the quality of a physical unclonable function using configurable ring oscillators," in *Proc. Int. Conf. Field Programmable Logic Appl., FPL'09*, Aug. 2, 2009, pp. 703–707.
[46] B. Badrignans, R. Elbaz, and L. Torres, "Secure FPGA configuration architecture preventing system downgrade," in *Proc. Int. Conf. Field Programmable Logic App., FPL'08*, Sep. 8–10, 2008, pp. 317–322.
[47] K. Kepa, F. Morgan, and K. Kosciuszkiewicz, "IP protection in partially reconfigurable FPGAs," in *Proc. Int. Conf. Field Programmable Logic Appl., FPL'09*, Sep. 2, 2009, pp. 403–409.
[48] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and Horowitz, "Architectural support for copy and tamper resistant software," in *SIGPLAN Not. 35,* 2000, pp. 168–177.
[49] D. Champagne, R. Elbaz, C. Gebotys, L. Torres, and R. B. Lee, "Forward-secure content distribution to reconfigurable hardware," in *Reconfigurable Computing and FPGAs,* 2008.
[50] J. Note and É. Rannaud, "From the bitstream to the netlist," in *Proc. 16th Int. ACM/SIGDA Symp. Field Programmable Gate Arrays*, Feb. 24–26, 2008, pp. 264–264.
[51] P. Athanas, J. Bowen, T. Dunham, C. Patterson, J. Rice, M. Shelburne, J. Suris, M. Bucciero, and J. Graf, "Wires on demand: Run-time communication synthesis for reconfigurable computing," in *Proc. Int. Conf. Field Programmable Logic Appl., FPL'07*, Aug. 27–29, 2007, pp. 513–516.
[52] R. Torrance, "The state-of-the-art in IC reverse engineering," in *Proc. Cryptographic Hardware and Embedded Systems (CHES 2009)*, vol. 5747, Lecture Notes in Computer Science, pp. 363–381.

Michael Gora received the M.S. degree in computer engineering from the Bradley Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University (Virginia Tech), Blacksburg, in 2010.

Currently, he is pursuing his professional career in industry. His interests include secure embedded systems, intellectual property protection, embedded security, and secure protocol development.

Abhranil Maiti received the B.E. degree in instrumentation engineering from Jadavpur University, Kolkata, India, in 2002, and the M.S. degree in electrical engineering from Illinois Institute of Technology, Chicago, in 2006. He is currently working towards the Ph.D. degree at the Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg.

His research interests include embedded system security and secure hardware design.

Patrick Schaumont (SM'06) received the M.S. degree in computer science from Rijksuniversiteit Ghent, Belgium, in 1990 and the Ph.D. degree in electrical engineering from the University of California, Los Angeles, in 2004.

He is an Assistant Professor of Computer Engineering at the Virginia Polytechnic Institute and State University (Virginia Tech), Blacksburg. He has been a Researcher at the Inter-university Micro-Electronics Center (IMEC), Belgium, from 1992 to 2001. He has served on the program committee of international conferences in this field such as CHES, DATE, DAC, IEEE HOST and IEEE MEMOCODE. His research interests include design of, and design methodologies for, secure embedded systems.

Dr. Schaumont served as Guest Editor for the IEEE *Design and Test Magazine*, ACM *Transactions on Reconfigurable Technology and Systems*, and the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS.