A Flexible Design Flow for Software IP Binding in Commodity FPGA

Michael A. Gora, Abhranil Maiti, Patrick Schaumont
Secure Embedded Systems, Bradley Department of Electrical & Computer Engineering
Virginia Polytechnic Institute and State University
Blacksburg Virginia, USA
gora@vt.edu, abhranil@vt.edu, schaum@vt.edu

***Abstract*— Software intellectual property (SWIP) is a critical component of increasingly complex FPGA based system on chip (SOC) designs. As a result, developers want to ensure that their SWIP sources are protected from being exposed to an unauthorized party and are restricted to run only on a trusted FPGA platform.**

This paper proposes a novel design flow for protecting SWIP by binding it to a specific FPGA platform. 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 to successfully protect a SWIP by encryption using a 128 bit FPGA-unique key extracted from a PUF.

Security Applications; Software Binding; Intellectual Property; Physical Unclonable Functions; FPGA

I. INTRODUCTION

Embedded systems and SOC designs based on FPGAs are becoming increasingly complex in nature, requiring sophisticated software development. As a result, developers need to protect their Software Intellectual Property (SWIP) in order to prevent theft or reverse engineering which can cause loss in revenue. This is supported by the fact that nearly 10% of the high technology products in the market are not genuine [3].

Hardware intellectual property (HWIP) in an FPGA-based system can be protected using bitstream encryption provided by FPGA manufacturers. However, providing security to the SWIP is not a readily available option. During the development and manufacturing stage, the SWIP can be considered secure assuming that the developer's environment is capable of protecting the SWIP source code from being exposed to other parties. Once sold to an end-user, the developer can no longer ensure that the SWIP is not misused or tampered with unless proper measures are adopted.

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 is deployed on a valid platform, which includes a valid (designated) FPGA and a valid (designated) HWIP. Figure 1 illustrates that the SWIP only functions correctly when a valid HWIP and a valid FPGA are present, such as with platform 2. Platform 3 fails because the FPGA device is not valid, while platform 1 fails because the HWIP is not valid. The identity of a design is thus formed by the combination of an FPGA and a HWIP. In this work, we propose an end-to-end design flow for binding a SWIP to a commodity FPGA.

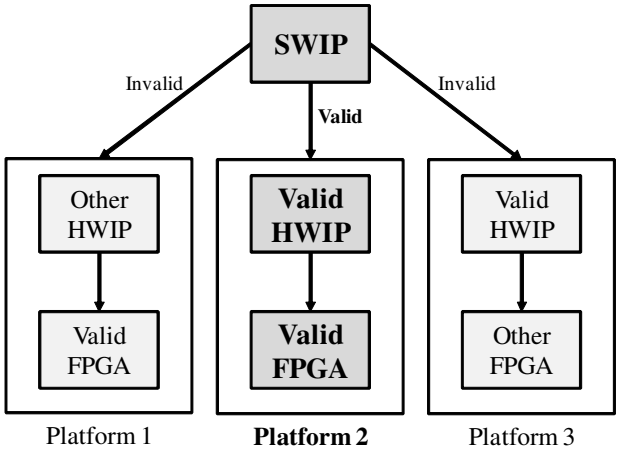

Figure 1. SWIP binding to a platform comprised of HWIP and an FPGA.

SWIP binding can be achieved using costly mechanisms such as secure ROM or flash memory to store FPGA specific cryptographic keys. This is not only expensive but also 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.

Our main contributions in this paper are as follows-

- To propose a complete, end-to-end design flow to bind a SWIP to an 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.

The remainder of this paper is organized as follows: Section 2 provides an overview of related work. Our proposed design flow is presented in Section 3. In Section 4, the supporting hardware architecture is described. We present a security analysis of our design flow in section 5. Our design metrics and results are evaluated in Section 6. Finally, we conclude the paper in Section 7.

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, 18, 19]. Water-marking 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 features in modern processors [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]. The AEGIS provides for a single-chip processor with built-in tamper resistance and detection as well dedicated cryptographic components. However, these hardware solutions are inflexible and may not meet all the needs of an embedded system.

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 non-volatile 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 robust 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 [12], a FPGA SWIP protection mechanism is proposed using SRAM-based PUF. The emphasis of that research is on the development of a secure protocol to authenticate intellectual property components. In contrast, we present a solution for secure runtime integration of SWIP. Also, we demonstrate a system prototype.

A modified Aegis architecture has been proposed in [13] for secure software execution using PUF. This 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 software IP to an FPGA platform irrespective of the class or vendor of the FPGA.

A. *Overview*

Our idea is introduced in the context of different phases in FPGA-based system design starting from the developer to the end-user. While the end-user only perceives a finished product, the developer has three distinct components to manage.

- The FPGA is the physical silicon that the other components are deployed on. It might contain a hard core processor or other specialized hardware such as multipliers.
- The HWIP represents 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].
- Finally, there is the SWIP that executes on a soft core or hard core processor in the FPGA. This software is often stored outside of the FPGA bitstream. We propose a design flow to bind it to an FPGA.

Figure 2 illustrates how we employ our PUF based SWIP binding methodology in the process of FPGA-based system design. The boxes in gray represent the major components of our method.

Our methodology is broadly divided in two parts as follows:

- Before delivery to the end-user, an FPGA-unique key is extracted using the PUF in a process called PUF enrollment. The SWIP is encrypted using the 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.

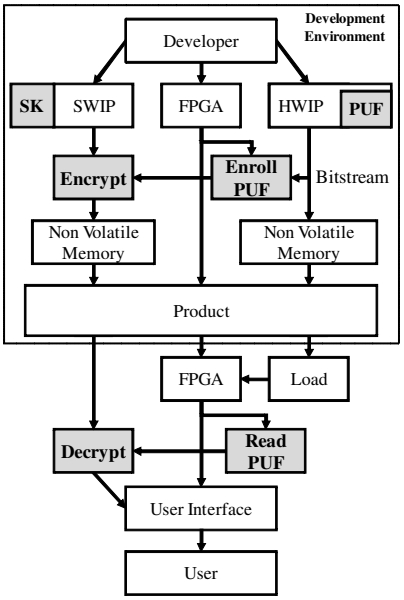

Figure 2. FPGA system design flow from developer to end-user.

The FPGA device, the hardware bitstream and the encrypted SWIP with the SK constitute the final product to the end-user. It is critical to prevent an attempt to modify any of the components of the delivered product for the purpose of finding out the PUF key. To solve this problem, we implemented an integrity mechanism which is discussed in detail in section 3.D.

B. SWIP Encryption

Ideally, all portions of the binary would be encrypted. 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. The C attribute functionality is used to specify the name for sections related to the SK which are labeled as '.secKern". All other sections, regardless of their content, are considered SWIP sections.

A plain text binary containing SWIP and our SK is depicted in Figure 3 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 Encryptor (IPE) utility that can extract and parse the header from the binary. The header contains information about the name, location, and size of the software in that particular section. Based on this information and our naming convention, the IPE retrieves the SWIP section from the binary file. Encryption is performed on this section through 128-bit AES in counter mode [4] using a key derived from the PUF.

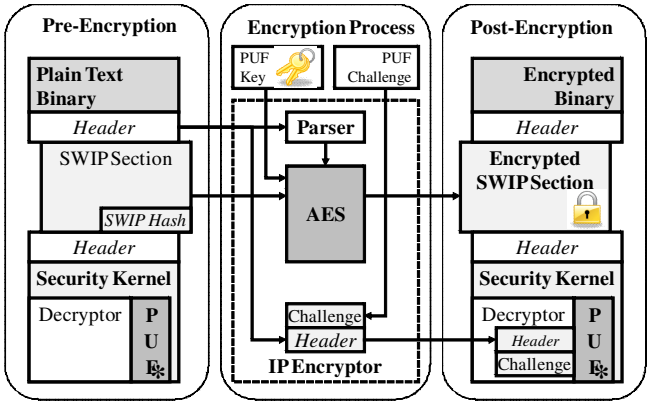

Figure 3. SWIP binding encryption process. PUF software itnerface is denoted by the asterix (*).

Additionally, the IPE inserts plain text information into the SK. This includes the PUF challenge, the header information, and a simple XOR hash of all the bytes in an SWIP section for subsequent use during decryption process. In the end, a binary is produced containing both the encrypted SWIP and the plain text SK section.

C. SWIP Decryption

When the encrypted binary is downloaded to an FPGA, the SK performs the operations of the IPE in reverse as shown in Figure 4. 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. To ensure an error free decryption, the SK performs a simple word wise XOR hash of the decrypted SWIP section. This value is compared to a hash performed by the developer and included at the end of each SWIP section. If the hash values differ this could indicate an invalid SWIP 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 freed for other uses.

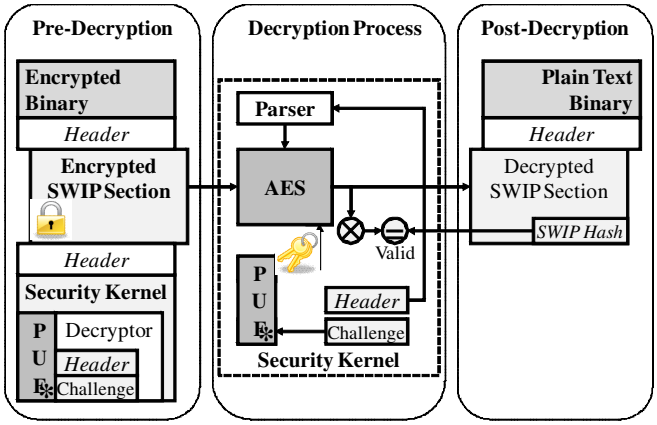

Figure 4. SWIP binding decryption process. PUF software itnerface is denoted by the asterix (*).

The drawback of this scheme is that the security kernel 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 an adversary could modify the security kernel and utilize it to retrieve the PUF key, and the decrypted SWIP. We address this issue with the inclusion of an Integrity Kernel (IK) as discussed in section III.D.

D. *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 entrusted software and extract the key from the PUF. To maintain the trusted boot procedure (Figure 5), we introduce an obfuscated ROM to provide a secure storage mechanism for the IK and all values needed to control the boot process.

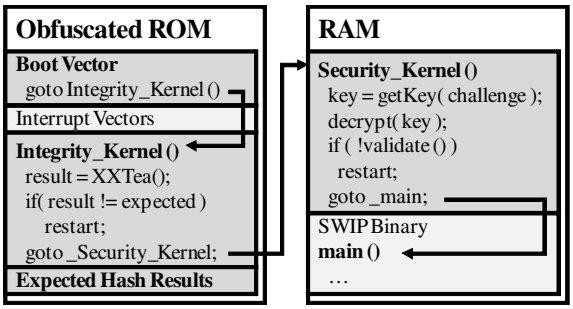

Figure 5. Trusted boot procedure.

1) *Obfuscated ROM*

An FPGA bitstream is believed to have an inherent layer of obfuscation. Though LUT configuration and BRAM contents can be accessed relatively easily, a complete reverse engineering of the bitstream into a true netlist has not been reported so far [14]. Based on this assumption, an obfuscated ROM is implemented using multi-level logic in the FPGA avoiding direct storage in LUT or BRAMs. This ROM is used to securely store the integrity-kernel binary. It should be noted that even though the 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 a result of the logic circuits being formed by a combination of several LUTs.

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 in the HDL source of the ROM. Since synthesizing the obfuscated ROM is easy, modifying the design of the integrity kernel is not difficult. These features help to maintain the flexibility of our design flow. The only major design concern is maintaining a low footprint which is determined by the size of the FPGA.

2) *Hashing Algorithem*

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 input blocks. An alternative solution is the use of a cipher based Davies-Mayer hash. Davies-Mayer hash 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 implement it in a 512 byte block of obfuscated ROM.

E. *Design Summary*

All the software and hardware components, required for our design flow, are illustrated in Figure 6 below from a developer's perspective with shaded figures as main components of our design flow.

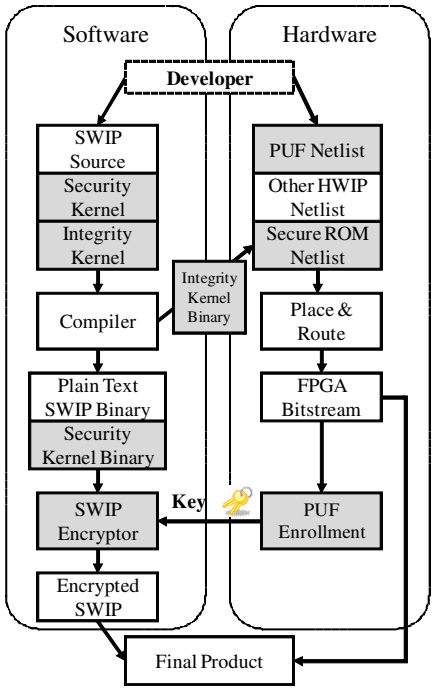

Figure 6. Detailed SWIP binding design flow.

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 adequate space in fabric and 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.

IV. PROTOTYPE SYSTEM

This section presents the underlying hardware architecture which has been used to build our prototype implementation. It consists of standard hardware components and the PUF.

A. FPGA Hardware Architecture

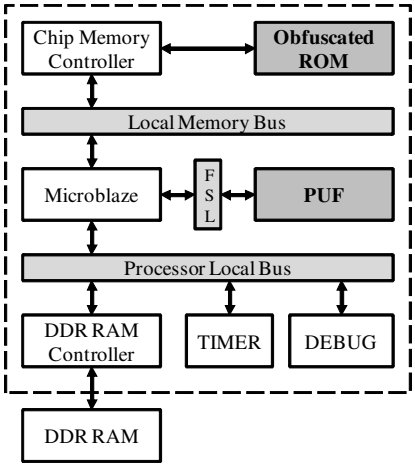

Figure 7. SWIP binding prototype system architecture on a Xilinx Spartan XC3500E FPGA.

Figure 7 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.D. Platform evaluation is further supported with a timer and a debug module. The dotted box in Figure 7 indicates the boundary of the FPGA. Anything outside it is an off-chip component, and is non-trusted.

B. Physical Unclonable Function (PUF)

For our prototype implementation, we used a ring-oscillator (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. This type of PUF is by far one of the easiest to create on an FPGA because it has very few routing constraints, and even small manufacturing variations yield a stable and unique PUF output.

1) PUF Deployment

We implemented the RO-based PUF on a Xilinx Spartan XC3S500E FPGA. To ensure that the RO frequency variations are determined by manufacturing variations (and not, for example, by routing variations), we implemented a five-stage ring oscillator as a hard macro as proposed in [1].

A C- program is used to control the PUF and extract the challenge/response pairs. The process of characterizing the PUF for the first time is called PUF enrollment. The encryption keys, required for SWIP encryption, are derived from the PUF enrollment.

2) Error Correction of PUF

When using a RO based PUF, there is a possibility that some bits in the response of the PUF produce varying results due to noise. A PUF error correction scheme can be used to prevent these errors, and several correction schemes have been proposed such as [10]. 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. Therefore we have implemented the following simple but effective error correction mechanism.

There are 256 ROs in the PUF and a 255 bit key is extracted by the following method –

$Ri = 1$ *if* $f_i > f_{i+1}$ *where* $i = 0$ *to* 254

$= 0$ *otherwise*

Here fi stands for the frequency of the i-th ring oscillator and Ri is the i-th bit of the key generated as a response from the PUF. It is clear that the stability of the response bit Ri depends on the difference in frequencies fi and $fi+1$. Higher difference will result in more stability. Based on this, we perform the following as error correction.

- We measure $\Delta f = | f_i - f_{i+1} |$ *for* $i = 0$ *to* 254 during enrollment of PUF.
- All Δf values are then sorted in decreasing order.
- The first 128 pairs of frequencies in the sorted list are used to build a 128 bit key required for encryption, and rest is ignored. We selected the first 128 bits because the encryption of the SWIP is done using a 128-bit key.

In this way, those pairs of ROs which have relatively higher difference in frequency are used to extract a stable key. The drawback of this scheme is that we are not able to fully utilize all the ring oscillators for key extraction.

V. SECURITY ANALYSIS

At first, we assume that the developer's environment will protect both HWIP and SWIP sources (dotted box in Figure 2 defines the developer's environment). 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, the complex and sensitive delay behavior of the PUF changes and the key is destroyed.

2) Other 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. Moreover, we avoided use of easily accessible components like LUTs and BRAMs to store sensitive information. As a result, it becomes extremely difficult to extract useful information from the bitstream or to modify it [14]. As long as these assumptions can be made concerning the nature of the FPGA bitstream, adequate security is available to bind SWIP without the need for a dedicated bitstream encryption scheme. Additionally, when combined with a FLASH based FPGA the ability of an attacker to modify the bitstream or extract the PUF is greatly reduced.

B. Software Analaysis

1) Software Integrity

As the SWIP is encrypted, it is very difficult to modify it in any useful way. On the other hand, since the security kernel is in plain text, it could be modified. However, the addition of the integrity kernel using the obfuscated ROM prevents the execution of the security kernel if it has been changed.

2) Execution of Malicious Code

We want to ensure that the integrity kernel and the security kernel execute first before any other code. This is accomplished by coding the entry point and the interrupt vectors into the obfuscated ROM. This prevents an attacker from moving execution outside of the secure code or interrupting the secure boot procedure.

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.

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 C-code binary was encrypted using each of these keys. All of these five encrypted binaries were downloaded on all five FPGAs separately.

Each FPGA allowed the execution of only one distinct binary which was encrypted using the PUF-key extracted from that particular FPGA. Table 1 shows the distinct keys generated from five FPGAs.

TABLE I. KEYS GENERATED FROM DIFFERENT FPGAS

FPGA	128 bit Key
FPGA 1	3EC71AE95B952BF38CCBFD6DB960D2A8
FPGA 2	62AF3E0CADF38B19C188E5AA2261A756
FPGA 3	37736357E1B7A2958D72DE12C537B509
FPGA 4	48D0D47D40D62D91B1B4A2446EADAF8C
FPGA 5	7C802AF6FF2C976027BDFBF46FE844FE

B. PUF Characterization

We measure two parameters namely uniqueness and reliability to characterize the PUF.

Uniqueness – 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.

If a PUF produces two n-bit responses *ri* and *rj* when implemented on two different FPGAs *i* and *j* respectively for a particular challenge *c*, then the hamming distance *hij* between *ri* and *rj* will determine if they are distinguishable or not. The difference between *ri* and *rj* is mainly caused by the inter chip variation due to variation in manufacturing process. If this variation follows a uniform random distribution, then the average value of *hij* would be around 50% of *n*.

Given a PUF is implemented on *k* FPGA chips, we define its uniqueness *U* as the average of the percentage hamming distance between the responses from every pair of implementations.

$$U = \frac{2}{k(k-1)} \sum_{i=1}^{i=k-1} \sum_{j=i+1}^{j=k} \frac{h_{ij}}{n} \times 100\% \qquad (1)$$

Reliability of PUF expresses the stability of a response r that is produced by a PUF from an FPGA for a challenge *c*. It is critical for a PUF to reproduce the same response with minimum rate of error for a particular challenge *c* if *c* is applied to the PUF multiple times.

Reliability, R = % of number of stable bits in a response produced by a PUF.

In this experiment, we estimated the uniqueness and the reliability under constant operating environment. The following table shows the values of the two metrics of the PUF that we implemented.

TABLE II. PUF UNIQUENESS AND RELIABILITY

Number of ROs	128 bit Key	Reliability
256	44%	96.7%

It can be observed that 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 (4.2.2).

TABLE III. PUF EXTRACTION TIME

Enrollment	90 seconds
Runtime Extraction	4 seconds

It can be observed in Table 3 that though the enrollment time period is long, which is necessary to generate a stable key, it is a onetime operation. The time required for runtime key extraction is much shorter.

C. *Overall Hardware Utilization*

Table 4 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.

TABLE IV. RESOURCE UTILIZATION ON XILINX SPARTAN XC3S500E FPGA

Components	LUTs	Slices
Microblaze	1397	698
PUF	931	466
Obfuscated ROM	559	279
DDR RAM Controller	1304	1024
All Components	4563	3678

D. *Security Software Overhead*

As evident from the Tables 5 and 6, 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 V. SECURITY KERNEL MEMORY REQUIRMENTS

Component	Size (Bytes)
AES	10560
Total	14720

TABLE VI. INTEGRITY KERNEL MEMORY REQUIRMENTS

Component	Size (Bytes)
Boot/Interrupt Vectors	80
XXTEA	376
Expected Hash Results	8
All Components	4563

VII. 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. Coupled with a flash based FPGA device this can be used as a stepping stone for solving other security problems in trusted computing on FPGAs.

ACKNOWLEDGMENT

The work reported in this paper was supported in part by the National Science Foundation Grant N0 0644070.

REFERENCES

[1] G. E. Suh and S. Devadas. Physical Unclonable Functions for Device Authentication and Secret Key Generation. In Proceedings of Design Automation Conference, June 2007.

[2] 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 International Symposium on Circuits and Systems, May 2008.

[4] N. Ferguson, B. Schneier. Practical Cryptography. Wiley Publishing Inc.

[5] Wheeler, D., Needham, R.: TEA extensions. Technical report, Cambridge University, England (October 1997).

[6] R. Winternitz. A secure one-way hash function built from DES. In Proceedings of the IEEE Symposium on Information Security and Privacy, p. 88-90. IEEE Press, 1984

[7] Xilinx Embedded Systems Tool Reference Manual.

[8] R. Anderson and M. Kuhn. Tamper Resistance – A Cautionary Note. In Proceedings of 2nd USENIX Workshop on Electronic Commerce, November 1996.

[9] S. P. Skorobogatov. Semi-Invasive Attacks - A New Approach to Hardware Security Analysis. In Technical Report UCAM-CL-TR-630. University of Cambridge

[10] C. Bosch, J Guajardo, A Sadeghi, J Shokrollahi, P. Tuyls. Efficient Helper Data Key Extractor on FPGAs. In Proceedings of the Cryptographic Hardware and Embedded Systems – CHES 2008.

[11] Xilinx. Inc., Using bitstream encryption," Handbook of the Virtex II Platform. [Online]. Available: http://www.xilinx.com

[12] J. Guajardo, S. S. Kumar, G.-J. Schrijen and P. Tuyls, FPGA intrinsic PUFs and their use for IP protection. In Cryptographic Hardware and Embedded Systems, 2007.

[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. In Proceedings of the 32nd Annual International Symposium on Computer Architecture, June 2005.

[14] Saar Drimer. Volatile FPGA design security – a sruvey. April, 2008. http://www.cl.cam.ac.uk/~sd410/papers/fpga_security.pdf

[15] Atallah, M. J., Bryant, E. D., Korb, J. T., and Rice, J. R. 2008. Binding software to specific native hardware in a VM environment: the puf challenge and opportunity. In Proceedings of the 1st ACM Workshop on Virtual Machine Security (Alexandria, Virginia, USA, October 27 - 27, 2008). VMSec '08. ACM, New York, NY, 45-48.

[16] Collberg, C.S.; Thomborson, C., Watermarking, tamper-proofing, and obfuscation - tools for software protection. In Software Engineering, IEEE Transactions on , 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. In Security and Privacy in Digital Rights Management, LNCS, 2320:141-159, 2002.

[18] H. Chang, M. Atallah. Protecting software code by guards. In Security and Privacy in Digital Rights Management, LNCS, 2320:160-175, 2002.

[19] H. Jin, G. Myles, and J. Lotspiech, Towards Better Software Tamper Resistance. In ISC 2005, LNCS 3650, pp. 417–430, 2005.

[20] Xuesong Zhang; Fengling He; Wanli Zuo, Hash Function Based Software Watermarking. In Advanced Software Engineering and Its Applications, 2008. ASEA 2008 , vol., no., pp.95-98, 13-15 Dec. 2008

[21] Palsberg, J.; Krishnaswamy, S.; Minseok Kwon; Ma, D.; Qiuyun Shao; Zhang, Y., Experience with software watermarking. In Computer Security Applications, 2000. ACSAC '00. 16th Annual Conference , vol., no., pp.308-316, Dec 2000

[22] Ge, J., Chaudhuri, S., and Tyagi, A. 2005. Control flow based obfuscation. In Proceedings of the 5th ACM Workshop on Digital Rights Management (Alexandria, VA, USA, November 07 - 07, 2005). DRM '05. ACM, New York, NY, 83-92.

[23] Michiels, W. and Gorissen, P. 2007. Mechanism for software tamper resistance: an application of white-box cryptography. In Proceedings of the 2007 ACM Workshop on Digital Rights Management (Alexandria, Virginia, USA, October 29 - 29, 2007). DRM '07. ACM, New York, NY, 82-89.

[24] P. van Oorschot, A. Somayaji, G. Wurster, Hardware-assisted circumvention of self-hashing software tamper resistance, In IEEE Trans on dependable and secure computing. 2(2):82-92, April-June 2005.

[25] Suh, G. E., Clarke, D., Gassend, B., van Dijk, M., and Devadas, S. 2003. AEGIS: architecture for tamper-evident and tamper-resistant processing. In Proceedings of the 17th Annual international Conference on Supercomputing (San Francisco, CA, USA, June 23 - 26, 2003). ICS '03. ACM, New York, NY, 160-171.