PUF ROKs: Generating Read-Once Keys from Physically Unclonable Functions (Extended Abstract)

Michael S. Kirkpatrick
Department of Computer Science
Purdue University
West Lafayette, IN, USA
mkirkpat@cs.purdue.edu

Elisa Bertino
Department of Computer Science
Purdue University
West Lafayette, IN, USA
bertino@cs.purdue.edu

Sam Kerr
Department of Computer Science
Purdue University
West Lafayette, IN, USA
stkerr@cs.purdue.edu

ABSTRACT

Cryptographers have proposed the notion of read-once keys (ROKs) as a beneficial tool for a number of applications, such as delegation of authority. The premise of ROKs is that the key is destroyed by the process of reading it, thus preventing subsequent accesses. While the idea and the applications are well-understood, the consensus among cryptographers is that ROKs cannot be produced by algorithmic processes alone. Rather, a trusted hardware mechanism is needed to support the destruction of the key.

In this work, we propose one such approach for using a hardware design to generate ROKs. Our approach is an application of physically unclonable functions (PUFs). PUFs use the intrinsic differences in hardware behavior to produce a random function that is unique to that hardware instance. Our design consists of incorporating the PUF in a feedback loop to make reading the key multiple times physically impossible.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Physical security, security*

General Terms

Security

Keywords

physically unclonable functions, PUFs, read-once keys, ROKs, cryptography, hardware, delegation, SoC, ASIC

1. INTRODUCTION

The term read-once keys (ROKs) describes the abstract notion that a cryptographic key can be read and used only once. While it seems intuitive that a trusted piece of software could be designed that immediately deletes a key after using it, such a scheme naïvely depends on the proper execution of the program. This approach could be easily circumvented by running the code within a debugging environment that halts execution of the code before the deletion occurs. That is, the notion of a ROK entails a stronger protection method wherein the process of reading the key inherently results in it immediate destruction.

ROKs could be applied in a number of interesting scenarios. One application could be to create one-time programs [7], which could be beneficial for protecting the intellectual property of a piece of software. A potential client could download a fully functional one-time program for evaluation before committing to a purchase.

The ability to generate ROKs in a controlled manner could lead to an extension where keys can be generated and used a multiple, but limited, number of times. For example, consider the use of ROKs to encrypt a public key pk. If an identical ROK can be generated twice, the owner of pk could first use the key to create $e_{ROK}(pk)$ (indicating the encryption of pk under the public key ROK). Later, an authorized party could create the ROK a second time to decrypt the key. Such a scheme could be used to delegate the authority to cryptographically sign documents.

While the notion of ROKs is a promising one, the disheartening consensus is that ROKs cannot be created through algorithmic processes alone. That is, we are aware of no work that has proposed a ROK design that relies solely on software. Instead, trusted hardware is required to guarantee the immediate destruction of the key.

In this paper, we propose the creation of ROKs using physically unclonable functions (PUFs) [5, 6]. PUFs are based on the intrinsic randomness that exists in hardware. As such, a robust PUF creates a mathematical function that is unique to each physical instance of a hardware design. That is, even if the same design is used for two devices, it is physically impossible to make their PUFs behave identically.

Our insight for the design of such "PUF ROKs" is to incorporate the PUF in a feedback loop for a system-on-chip (SoC) design.[1] That is, our design is for the PUF to reside on the same chip as the processor core that performs the encryption. This integration of the PUF and the processor core protects the secrecy of the key. Attempts to read the key from memory or by eavesdropping on bus communication inherently fail, as the key never exists in either location.

[1]Our design could also be made to work for application-specific integrated circuits (ASICs), but we limit our discussion to SoC designs for simplicity.

CSIIRW'10, April 21-23, Oak Ridge, Tennessee, USA.

The random nature of PUFs ensures that each iteration of ROK computation will produce a unique, unpredictable key. However, providing the same initial seed value to the PUF will produce the same string of keys. Hence, *Alice* could provide an initial seed to produce a sequence of keys that are used to encrypt a set of secrets. *Alice* could then reset the seed value before sending the device to *Bob*. *Bob*, then, could use the PUF to recreate the keys in order, decrypting the secrets. As *Bob* has no knowledge of the seed value, he is unable to reset the device and cannot recreate the key just used.

2. PUFS

Before we describe our design for PUF ROKs, this section provides some relevant background on the creation, properties, and applications of PUFs. Research on PUFs [5, 6] arose from the observation that distinct instances of hardware produce unique behavioral characteristics [11]. That is, each copy of the device, even if designed to be identical, will exhibit slight variations that can be measured only during execution of the circuit. The precise behavior that ensues can be neither controlled nor predicted. In addition to silicon-based circuits, similar distinguishing techniques have been applied to RFID devices [3, 2].

Mathematically, a PUF can be modeled as a function $PUF : C \rightarrow R$, where C denotes a set of input challenges (usually encoded as a bit string) and R is a set of responses. That is, for a single device, providing the same input C_i to the PUF will yield approximately the same result R_i. In practice, the PUF output consists of noisy data; error-correcting codes, such as Reed-Solomon [12], can be applied to ensure that the response is identical every time.

The definition of the function is determined exclusively by the variations in the hardware. As a result, no properties can be assumed about the function itself. That is, in general, PUFs are neither linear nor injective nor surjective. Rather, the function merely consists of a set of random pairs (C_i, R_i). Furthermore, as the function is defined by the hardware, providing the same C_i as input to a different device's PUF will produce a different response $R_i' \neq R_i$.

As an example of a circuit-based PUF, consider the design in Figure 1. In this design, the one-bit challenge input switches which oscillator's output gets directed into which counter. After a certain amount of time, the counters are compared and an output bit reports which counter holds a larger value. While this design produces only a single bit of output, larger PUFs can generate longer bit strings.

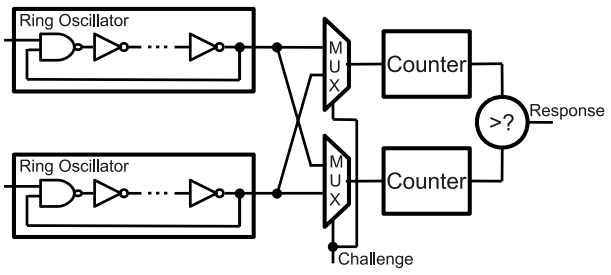

Figure 1: A sample 1-bit PUF based on ring oscillators

In this PUF, the ring oscillators are designed to be identical. However, as a result of the manufacturing process, the wire length and width will inevitably differ. Hence, one oscillator will switch between outputting a 1 and a 0 at a faster rate. But without actually executing the circuit, it is impossible to tell which oscillates faster. This is due to the fact that the difference between the two oscillators is too small to be measured. Thus, one cannot control or predict the output of this PUF just by inspecting the device.

PUFs have been applied in a number of settings. One common approach is to use the response to provide secure cryptographic key storage [9, 8]. For instance, to protect the key K, one could compute $X = K \oplus R_i$, where $\oplus$ denotes the bitwise XOR operator. As R_i is a random bit string, it acts as a one-time pad and the value X can then be stored in plaintext without sacrificing the confidentiality of K.

Other applications have also been proposed. One technique uses PUFs to bind software to hardware in a VM environment [1]. The AEGIS secure processor [15, 14] incorporates a PUF for key generation and storage. Robust PUFs have been proposed as an authentication scheme for banking environments [4]. Finally, in previous work, we have proposed the use of PUFs to bind authentication to trusted devices to combat insider threats [10].

3. PUF ROKS

In this work, we propose the use of PUFs to generate ROKs, which we call PUF ROKs. Like previous work [13], our design is based on the idea of using the PUF output to generate a transient key dynamically. We start this section by describing how to create a PUF ROK. We then describe a number of application scenarios and how PUF ROKs achieve the desired goals.

3.1 PUF ROK Design

Our SoC architecture for generating PUF ROKs is shown in Figure 2. The main Processor Core (PC) defines the primary functionality that is exposed to software. When the PC needs access to a PUF ROK, it issues a command to the Crypto Core (CC). The CC, then, is responsible for communicating with the PUF and performing the cryptographic operations.

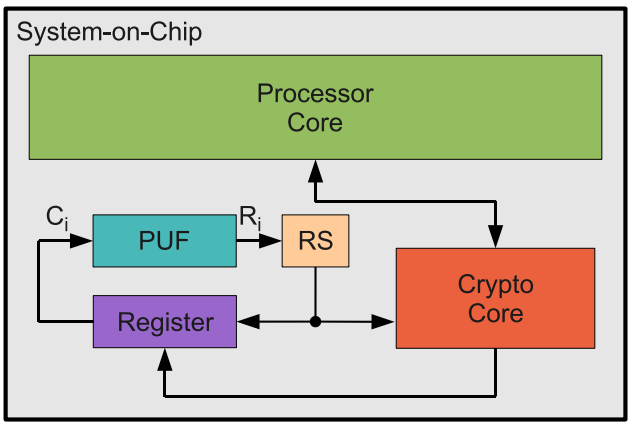

Figure 2: Components for a SoC PUF ROK design

The PUF is connected to a Register (Reg) as part of a

feedback loop. The PUF reads the value from the Reg and uses that value as its input challenge C_i. The PUF response R_i is then written back into the Reg and reported to the CC. For the sake of simplicity, we are assuming the PUF challenges and responses are the same length. We also assume that Reg consists of a small storage cache of the same size.[2]

3.2 Symmetric Key PUF ROKs

The functionality of the CC depends on the cryptographic application. As a first approach, consider using PUF ROKs for symmetric key cryptography. In this scenario, the PC issues the command $\mathsf{Enc}_{ROK}(m)$, where m indicates the message to be encrypted. The CC then issues the command $\mathsf{Set}(x)$, which writes the value x into the Reg.[3] The PUF then uses x as C_i and generates R_i, which is reported back to the CC.

To convert R_i into a usable key, the CC must perform two feats. First, it must create a set of error correcting codes for R_i. Recall that PUFs are built on mechanisms that involve noisy data. As such, when the PUF ROK is needed for decryption later, the error correcting codes allow the CC to recreate the key without a single bit error.

Next, to guarantee a strong key from the cleansed R_i, the CC applies a cryptographic hash. That is, to generate a 256-bit AES key, the CC computes $K = H(R_i)$, where H is the SHA-256 hash function. The purpose of the hash is to prevent the key from revealing information about the PUF itself. That is, if an attacker learns the key by observing the ciphertext and plaintext, the attacker cannot use this information to model the PUF. In addition, if R_i and R_j differ by only a single bit, the keys K_i and K_j will have a Hamming difference of 128 bits on average.

Once the CC has polled the PUF and generated the key, the encrypted message $e_{ROK}(m)$ is provided to the PC. Later, when the recipient wishes to decrypt the message (which can only be done once), the PC issues the command $\mathsf{Dec}_{ROK}(e_{ROK}(m))$ to the CC. The CC then resets the Reg with $\mathsf{Set}(x)$, and polls the PUF to recreate the key. The decrypted message, then, is returned to the PC.

While we have described the process for generating a single symmetric key K, our design allows the ordered creation and usage of a series of keys. For instance, let the CC perform the initialization $\mathsf{Set}(x)$ once. The CC can then generate a sequence of keys $K_1, K_2, \ldots, K_n$ by repeatedly polling the PUF. The decryption process, similarly, can be triggered by the same initialization. This technique is similar to using a chain of hash results $H(x), H(H(x)), H(H(H(x)))$, etc., to generate a sequence of keys. The advantage of the PUF ROK approach is that messages can only be decrypted in the same order that they were encrypted.

3.3 Public Key PUF ROKs

Now consider the case of public key cryptography. In this setting, we start with the assumption that the CC contains the necessary parameters for the public key computations. For instance, if the RSA cryptosystem is used, the CC knows (or can create) the two large prime numbers p and q such that $n = pq$. The goal, then, is to generate a pair (pk, sk), where pk denotes a public key and sk denotes the corresponding private key.

In contrast to the symmetric key approach, the CC does not need to generate the ROK twice. As such, the $\mathsf{Set}(x)$ function is optional. However, the CC still polls the PUF to generate the pair of keys. The challenge with using a PUF to create a public key pair, though, is how to generate a bit string that is long enough. A strong RSA key, for example, is at least 2048 bits long. But creating a 2048-bit PUF output would require a prohibitively large circuit design.

Instead, our approach is for the CC to buffer a series of PUF results. For instance, if the PUF produces a 256-bit output, the CC could use R_i as bits 0-255, R_{i+1} as bits 255-511, and so forth. Once the CC has polled the PUF to get a sufficient number of random bits, the next challenge is to convert this bit string into a strong key. For simplicity, we assume the use of RSA.

Let e denote the candidate key that the CC has received from the PUF. In order to use e as an RSA key, e must be coprime to $\varphi(n) = (p-1)*(q-1)$. By applying the Euclidean algorithm, the CC can compute the greatest common divisor $g = gcd(e, \varphi(n))$. If $g = 1$, e and $\varphi(n)$ are coprime, and e can be used as is. Otherwise, $e' = e/g$ can be used. The secret key sk, then becomes e or e' as appropriate. To compute the public key pk, the CC computes the modular multiplicative inverse of sk by using the extended Euclidean algorithm. That is, the CC computes d such that $sk * d \equiv 1 \pmod{\varphi(n)}$. This value d then becomes the public key pk.

Given this key pair (pk, sk), the PUF ROK can be used by the PC in multiple ways. First, the PC could issue the command $\mathsf{Sign}_{ROK}(m)$ to the CC, requesting a cryptographic signature. After generating (pk, sk), the CC uses sk to sign m, returning the signature and the public key pk to PC. pk can then be used by a third party to verify the signature.

Alternatively, the PC could issue the command Gen_{ROK}, which tells the CC to generate the key pair. Instead of using the keys immediately, the CC stores sk and returns pk to the PC. A third party wishing to send an encrypted message to the PC could use pk as needed. Then, the PC would issue Dec_{ROK} to have the CC decrypt the message.

Finally, consider the case where the third party needs assurance that the public key pk did, in fact, origin from the PUF ROK. This can be accomplished if the CC contains a persistent public key pair, similar to the Endorsement Key (EK) stored in a Trusted Platform Module (TPM). In addition to providing the pk to the PC, the CC could also return $\mathsf{Sign}_{EK}(pk)$, denoting the signature of the pk under this persistent key. This technique provides the necessary assurance, as the persistent key is bound to the CC.

3.4 Applications

Now consider the applications of PUF ROKs. Goldwasser *et al.* [7] proposed a technique for one-time programs, based on the assumption that the application is encrypted under a one-time use key. Closely related to one-time programs are delegated signatures. If *Alice* has a persistent key sk_A, she could encrypt this key with the PUF ROK as $e_{ROK}(sk_A)$. *Bob* would then provide this encrypted key to the PUF ROK, which decrypts it and uses the decrypted key to sign a single document on *Alice*'s behalf.

[2]Depending on the size of the PUF output, Reg may correspond to an array of hardware registers. *E.g.,* if the PUF output is 256 bits and hardware registers are only 32 bits, then Reg consists of 8 physical registers.

[3]We explore the generation of x in the full version of the paper, and omit this discussion from this abstract for space.

In homage to the late Peter Graves[4], we propose the use of PUF ROKs for self-destructing messages. If *Alice* has a portable PUF ROK device, she could use it to generate $\mathsf{Enc}_{ROK}(m)$. After receiving the message, *Bob* could use the device to decrypt the message. Once this is done, repeated attempts to decrypt the message would fail, as the Reg would no longer store the necessary challenge input.

Finally, consider the scenario of usage control. In this case, *Bob* has a public key PUF ROK device that contains the TPM-like key EK. *Bob* could use the device to retrieve the signed *pk*, which he sends to *Alice*. *Alice*, after confirming the signature, uses the key to encrypt the protected resource, sending the result to *Bob*. *Bob* can then use the *sk* stored on the PUF ROK to access the resource. Once the CC uses the key, the key is no longer accessible, and access to the resource is revoked.

4. CONCLUSION

In this work, we have proposed the use of PUFs to generate ROKs. The basic idea is to combine the PUF with a register that creates a feedback loop. The result is that no data required for the PUF ROK ever exists outside of the processor itself. By preventing the exposure of the key outside of the processor, an adversary is unable to read and store the key. That is, the key can only be used once and is immediately destroyed.

5. REFERENCES

[1] 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 *VMSEC '08*. ACM, 2008.

[2] B. Danev, T. S. Heydt-Benjamin, and S. Čapkun. Physical-layer identification of RFID devices. In *Proceedings of the USENIX Security Symposium*, 2009.

[3] S. Devadas, E. Suh, S. Paral, R. Sowell, T. Ziola, and V. Khandelwal. Design and implementation of PUF-based "unclonable" RFID ICs for anti-counterfeiting and security applications. In *2008 IEEE International Conference on RFID*, pages 58–64, 2008.

[4] K. B. Frikken, M. Blanton, and M. J. Atallah. Robust authentication using physically unclonable functions. In *Information Security Conference (ISC)*, September 2009.

[5] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas. Controlled physical random functions. In *Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC)*, 2002.

[6] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas. Silicon physical random functions. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS '02)*, 2002.

[7] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. One-time programs. In *CRYPTO 2008*, pages 39–56, 2008.

[4]Peter Graves, who passed away during the writing of this paper, starred as James Phelps on the television show, "Mission: Impossible." On the show, secret agents received assignments via a small device that self-destructed after the message was played.

[8] J. Guajardo, S. S. Kumar, G.-J. Schrijen, and P. Tuyls. FPGA intrinsic PUFs and their use for IP protection. In *Proceedings of the 9th Cryptographic Hardware and Embedded Systems Workshop (CHES)*, pages 63–80, 2007.

[9] J. Guajardo, S. S. Kumar, G.-J. Schrijen, and P. Tuyls. Physical unclonable functions and public-key crypto for FPGA IP protection. In *International Conference on Field Programmable Logic and Applications*, pages 189–195, 2007.

[10] M. Kirkpatrick and E. Bertino. Physically restricted authentication with trusted hardware. In *The Fourth Annual Workshop on Scalable Trusted Computing (ACM STC '09)*, November 2009.

[11] K. Lofstrom, W. Daasch, and D. Taylor. IC identification circuit using device mismatch. In *Solid-State Circuits Conference, 2000. Digest of Technical Papers. ISSCC. 2000 IEEE International*, pages 372–373, 2000.

[12] M. Riley and I. Richardson. Reed-solomon codes. `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/pscico-guyb/realworld/www/reedsolomon/reed_solomon_codes.html`, 1998.

[13] G. E. Suh and S. Devadas. Physcal unclonable functions for device authentication and secret key generation. In *Proceedings of the 44th IEEE Design Automation Conference (DAC)*, pages 9–14. IEEE Press, 2007.

[14] G. E. Suh, C. W. O'Donnell, and S. Devadas. AEGIS: A single-chip secure processor. In *Elsevier Information Security Technical Report*, volume 10, pages 63–73, 2005.

[15] G. E. Suh, C. W. O'Donnell, and S. Devadas. Aegis: A single-chip secure processor. *IEEE Design and Test of Computers*, 24(6):570–580, 2007.