Logically Reconfigurable PUFs: Memory-Based Secure Key Storage

Ilze Eichhorn
Intrinsic-ID
High Tech Campus 9
Eindhoven, The Netherlands
ilze.eichhorn@intrinsic-id.com

Patrick Koeberl
Intel Ireland
Collinstown Industrial Park
Leixlip, Ireland
patrickx.koeberl@intel.com

Vincent van der Leest
Intrinsic-ID
High Tech Campus 9
Eindhoven, The Netherlands
vincent.van.der.leest@intrinsic-id.com

ABSTRACT

The security of hardware is essential to the prevention of cloning, theft of service and tampering, and therefore to revenue preservation. An important component of hardware security is secure key storage. The level of security provided by a key is dependent upon the effort required from an attacker to compromise the key. Since the sophistication of tools used to carry out such attacks has increased significantly, protection of traditional key storage approaches, like storing a key in non-volatile memory (NVM), decreases. To fight these attacks Hardware Intrinsic Security (HIS) can be used. An example of HIS are Physically Unclonable Functions (PUFs). In this paper we introduce a new logically reconfigurable PUF (LR-PUF), based on a memory-based PUF. This LR-PUF uses the physical properties of a PUF combined with state information that is stored in NVM. Even though this implementation requires NVM, we will prove that the LR-PUF provides significantly more security than simply storing a key in NVM. The reason for this is that reading the information in NVM will not allow an attacker to derive any information on the key.

Categories and Subject Descriptors

B.7.m [**Hardware**]: Integrated Circuits—*Miscellaneous*

General Terms

Design, Security

1. INTRODUCTION

PUFs are a promising security primitive that utilize small physical differences between integrated circuits (ICs). These differences are due to deep-submicron manufacturing process variations. Because of these process variations, every transistor in an IC has slightly different physical properties that lead to measurable differences in terms of its electronic properties, like threshold voltage and gain. Given the fact that these process variations are uncontrollable during manufacturing, the physical properties of a device cannot be copied or cloned. Based on these properties an electronic "fingerprint" can be derived for each specific device, which is easy to measure but almost impossible and economically not viable to clone.

STC'11, October 17, 2011, Chicago, Illinois, USA.

1.1 Related work

Pappu [13] introduced the concept of PUFs in 2001 using the name Physical One-Way Functions. The proposed technology was based on obtaining a response (scattering pattern) when shining a laser on a bubble-filled transparent epoxy wafer. In 2002 this principle was translated by Gassend et al. [3] into Silicon Physical Random Functions. These functions make use of the manufacturing process variations in ICs, with identical masks, to uniquely characterize each IC based on oscillation frequencies (Ring Oscillator PUFs). In 2004 Lee et al. [8] proposed another PUF that is based on delay measurements, the Arbiter PUF.

Besides these PUFs based on delay measurements a second hardware intrinsic type is known: the memory-based PUF. These PUFs are based on the measurement of start-up values of memory cells. This memory-based PUF type includes SRAM PUFs (Guajardo et al. [4] in 2007), Butterfly PUFs (Kumar et al. [6] in 2008), and D flip-flop PUFs (Maes et al. [10] in 2008).

Reconfigurable PUFs (R-PUFs) have been introduced by Kursawe et. al. in [7]. Previous work on PUFs mainly considered static challenge-response PUFs. However, in many applications a dynamic PUF would be desirable. The R-PUF is a PUF with a mechanism that is able to transform it when required into a new PUF with a new unpredictable and uncontrollable challenge-response behaviour. However, a problem with the R-PUFs that are described in [7] is that none of them seem to be implementable in a practical way for security purposes. In [5] Katzenbeisser et. al. introduce the concept of Logically Reconfigurable PUFs (LR-PUFs). These LR-PUFs are based on a combination of physical properties of a regular (static) PUF and state information in NVM. LR-PUFs can be dynamically "reconfigured" by updating the state information, such that their challenge-response behaviour changes in a random manner.

1.2 Secure key storage

One of the fundamental uses of PUFs is secure key storage [14, 9]. Using PUFs for this purpose eliminates the need for storing cryptographic keys somewhere in the device in

non-volatile memory (NVM). When stored in NVM, keys are always present in the device even when it is switched off. Therefore, attackers can easily subject those devices to physical attacks using a variety of tools in order to get access to the secret keys. Using PUFs, the key is no longer stored in the device in a digital form. It is only constructed when it is needed. This limits the possibilities for an attacker immensely, since the time window during which a key is present is minimal and only occurs when power is on.

The type of PUFs that are generally used for secure key storage are memory-based PUFs. This subgroup of PUFs is described in section 1.1. There are two main reasons for using this subgroup of PUFs for secure key storage. The first reason is that most memory-based PUFs use components (like SRAM, Flip-flops) that are present in almost any device today. Secondly, the measuring circuits for these PUFs are very simple and can be designed without specific back-end design constraints. Therefore, costs are limited in comparison to PUFs that are based on special delay measuring circuitry (like arbiter and ring-oscillator PUFs), which require additional attention during the back-end design process.

When using PUFs for secure key storage, an important requirement is to derive robust cryptographic keys from noisy measurement data. To be able to do this, a helper data algorithm referred to as a "Fuzzy Extractor" [2] will be used. Generally, the main functions of a Fuzzy Extractor (FE) are error correction to remove noise and privacy enhancement to compress a string into a full entropy key. In the proposed LR-PUF construction (see section 4), privacy enhancement is performed by a separate block (the Hash). Therefore, the FE will only be used for error correction. How the FE is used in this implementation can be found in section 4.2.

1.3 Our contribution

This paper contains the first ever construction of logically reconfigurable PUFs (as introduced in [5]) created using memory-based PUFs. This LR-PUF is specifically suitable for hardware/software-binding solutions, which require the added functionality of protection against tampering with software versions. The main focus of the paper is to propose a secure hardware implementation of this LR-PUF. To achieve this an attacker model is created, architecture and protocol concepts are proposed, design considerations are taken into account and resource estimations are made.

1.4 Paper outline

The paper is organised in the following way: section 2 contains the description of the use case, which is driving the development of the memory-based LR-PUF. An attacker model for secure key storage with this LR-PUF can be found in section 3. In section 4 the construction of the LR-PUF and proposed protocols are described. Extra considerations for (secure) design are noted in section 5. Section 6 contains an analysis to demonstrate how secure the implementation is against the proposed attacker model. After this analysis, the final conclusions are drawn in section 7.

2. USE CASE

In many of today's electronic products, embedded software plays an important role. Crucial parts are implemented in hardware, but more and more higher layers (drivers, user interface, etc.) are implemented in embedded software or firmware that is running on an internal processor or microcontroller to keep the design flexible.

An advantage of this is that it is possible to use software for device differentiation. The functionality of a device can be changed by updating the software (which can be done in the field). The opposite is also possible, a client could change to a subscription with less features (which might be cheaper). The product differentiation feature can however be abused by hackers, trying to upgrade devices without paying by simply copying the software.

Besides using software versions for product differentiation, they can also be used to achieve shorter design cycles since bugs can easily be repaired in later software upgrades. Changing software is easier and less costly than making new hardware. Hackers might try to downgrade software to previous versions with certain security holes or with previously available features that can be used to their advantage.

Solutions exist in which software is bound to specific hardware by encrypting software with a unique key permanently stored in NVM. Only the device that has the correct key is able to decrypt the software. Copying the software to a second device will not work since the cryptographic key is different on the second device. There are however some problems with these solutions:

1. **Cloning:** Devices can be cloned if the cryptographic key can be copied to another system. Therefore the key must be stored in a relatively expensive secure NVM.
2. **Downgrading:** Even with a safely stored key, an attacker can downgrade a system if all software versions of a device are encrypted with the same key.

In order to solve these problems we propose a new type of logically reconfigurable PUF (LR-PUF). This new type deals with the two problems listed above in the following way:

1. LR-PUFs can be used to securely store keys on a device in a cost-effective way. As described in section 1, using any type of intrinsic PUF makes the key dependent on device unique characteristics. Hence, copying the LR-PUF circuitry to another device will result in a different generated key. In this way it is possible to bind software securely to a specific device by encrypting software with the device's unique cryptographic PUF key, thereby preventing illegal feature upgrading.
2. To prevent downgrading of software, reconfigurability of the LR-PUF is used. When a new software version is distributed, the LR-PUF is reconfigured. After reconfiguration the LR-PUF has a new and unpredictable response. This results in a renewed cryptographic key. If the previous software is copied back to the device, it will not run since the cryptographic key has changed.

Note that it is important that the reconfiguration cannot be undone. In other words, reconfiguration has to be one-way.

3. ATTACKER MODEL

We first state our assumptions. The details of the LR-PUF algorithms are not secret. At the LR-PUF level it is assumed that an attacker cannot write an arbitrary state to the NVM and is prevented from reading the current state through any functional interface. The PUF primitive underlying the LR-PUF construction is assumed to be *physically*

uncloneable. A PUF is physically uncloneable if it is practically infeasible to create a new PUF instance with an identical challenge-response behaviour to the original. Moreover, the PUF primitive is assumed to be unpredictable, i.e., it is not possible to predict the PUF response with more than negligible probability. For the memory-based PUFs under consideration it is reasonable to assume that both properties hold. Note that memory-based PUFs are not *mathematically* uncloneable [11]. If the response of a memory-based PUF can be extracted from the LR-PUF, it will be possible to emulate its behaviour in software or with a suitable circuit.

At the system-level it is assumed that the LR-PUF is embedded within a System on Chip (SOC). This includes the security algorithms using the stored secret and the consumer of data which might, for example, be an embedded CPU. Furthermore, it is assumed that steps have been taken to prevent attacks on the LR-PUF and supporting symmetric cipher by non-invasive means, i.e., by exploiting design flaws, protocol flaws, inducing faults or monitoring side-channels.

If the system-level assumptions above hold, an attacker must resort to invasive attack methods. For modern deep-submicron technology nodes in particular, invasive attacks require expensive equipment and specialised skill sets placing such attacks out of the reach of all but well-resourced attackers. We distinguish between *static* and *dynamic* invasive attacks. Static invasive attacks are performed on powered-off devices. Static attacks may be destructive, for example exposing the active layer of a device by removal of all overlying metal interconnects. Dynamic attacks, on the other hand, are performed on functioning, powered-up devices and must not compromise the functionality of the device, at least for the duration of the attack.

In traditional NVM-based key storage applications static attacks may be successful since the key is present on the device even when powered off. In contrast, any practical attack on an LR-PUF based secure key storage must be dynamic, since the PUF-derived secret is not present in any digital sense on the powered off device. In effect, the hardware "attack surface" is reduced; static attacks are ruled out with a consequent increase in the resistance to invasive attacks.

As outlined in section 2 an attacker is likely to attempt two attack types on the LR-PUF based secure key storage. The first is a cloning attack which, if successful, allows an attacker to copy the protected asset (here: software) from the target system to the cloned system. The second is a downgrading attack where the attacker attempts to revert back to a previous software version on the same system. These two attacks will be explored in more detail in section 6.

4. LR-PUF DESIGN

4.1 LR-PUF construction

The principle of the memory-based LR-PUF is equivalent to the concept presented in [5]. The output of an LR-PUF is determined by a combination of the physical properties of a PUF and state information as maintained by central control logic. The response (output) behaviour of the LR-PUF, specific for all possible challenges (input), can be changed by updating this state information (reconfiguration).

Figure 1 shows the construction selected for the memory-based LR-PUF. The input is translated into a challenge for the physical PUF. In case of memory-based PUFs this challenge usually consists of turning the power of the memory off and on again, after which the start-up pattern of the memory can be read. Using a Fuzzy Extractor and helper data, error correction is performed on the measured noisy response to retrieve the exact same response as has been enrolled. After this, the corrected PUF response can be hashed with current state information from NVM to produce the LR-PUF output. To change the output behaviour of the LR-PUF, a reconfiguration protocol updates the current state to a new (random) state when required. By writing a new value to the NVM containing the state, the input of the hash (and thus the output of the LR-PUF) changes.

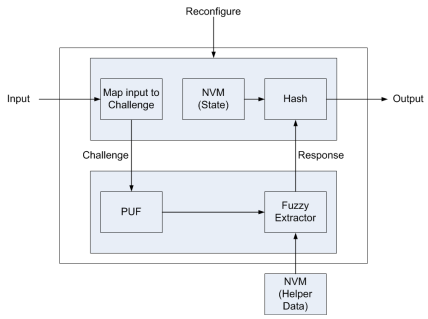

Figure 1: LR-PUFs: Memory-based construction

More details on the three phases of the system (*Enrolment*, *Reconstruction*, and *Reconfiguration*) can be found in section 4.2.

4.2 Protocol concepts

The memory-based LF-PUF has to be able to securely execute three different phases. For these phases, protocol concepts are developed: *Enrolment*, *Reconstruction*, and *Reconfiguration*. *Reconfiguration* is a rather new phase, which was introduced in [5]. We define it for memory-based PUFs.

4.2.1 Enrolment

Enrolment of an LR-PUF is required before a client device can be used in the field. This is performed in a secure environment, initialized by an authorized server. For enrolment the following parameters are involved:

r: response of PUF of client device
ID: identifier, random number generated by server
w: helper data
S_0: initial state, randomly generated by server
S_1: first state to be used for key generation
K_1: first generated symmetric key

Now the enrolment protocol, as can be seen in Figure 2, will be described. First on the server side two random parameters are generated: an initial state S_0 and an identifier ID. Using a function Hash, the first state is derived from the initial state: $S_1 \leftarrow \mathsf{Hash}(S_0)$. From this first state and the identifier, the first symmetric key is derived: $K_1 \leftarrow \mathsf{Hash}(ID|S_1)$. The server can now encrypt the first version of the software, SW_1, with the generated key K_1 and send $E_{K1}(SW_1)$ to the client, together with S_0 and ID.

Using the response r from the PUF on the client device, the helper data w is generated by the function Gen: $w \leftarrow \mathsf{Gen}(r, ID)$. On the server side ID, S_1 and K_1 are stored. In the client device S_1, w and the encrypted software are

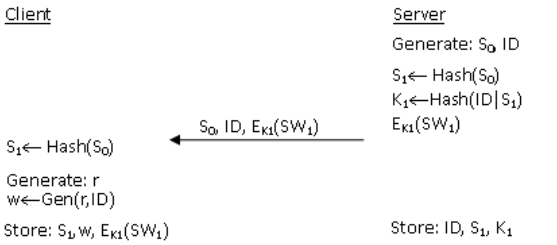

Figure 2: Enrolment protocol concept

stored in NVM. Now the client can decrypt this software using the key K_1 which it can retrieve by itself by executing the reconstruction protocol.

4.2.2 Reconstruction

In order for a client to run a software version SW_x that corresponds to the current state S_x, reconstruction of the key K_x is required, using the following parameters:

r': a noisy PUF response in client device
w: helper data
ID: identifier, random number generated by server
S_x: state to be used for key generation of K_x
K_x: generated symmetric key

By challenging the PUF in the client device, a noisy response r' is retrieved, related to the original response r by Hamming distance smaller than a given δ. The function Rcst reconstructs the key identifier: $ID \leftarrow \mathsf{Rcst}(r', w)$. Then the key can be generated again by the hash function: $K_x \leftarrow \mathsf{Hash}(ID|S_x)$. Since only state S_x is present in the client device, only key K_x can be reconstructed and only software version SW_x can be run.

4.2.3 Reconfiguration

In case of a software update, the system can be completely reconfigured for usage of a new key that only corresponds to the new software by performing the reconfiguration protocol, see Figure 3. To upgrade the software from SW_x to SW_{x+1}, key K_x has to be upgraded to key K_{x+1} which has to be present on both the client as the server. For this the following parameters are needed:

r': a noisy PUF response in client device
w: helper data
ID: identifier, random number generated by server
S_x: state to be used for key generation
K_x: generated symmetric key

The server initializes reconfiguration by a reconfigure command encrypted with current key K_x. The client receives a command, performs reconstruction using S_x to generate K_x and decrypts the command. Then the client reconfigures S_x using Hash: $S_{x+1} \leftarrow \mathsf{Hash}(S_x)$. Now reconstruction is performed again, using S_{x+1} to generate the new symmetric key K_{x+1}: $K_{x+1} \leftarrow \mathsf{Hash}(ID|S_{x+1})$. The client sends a response to the server indicating that reconfiguration was done successfully and encrypts this response with K_{x+1}. Since the server should have been able to perform the same reconstruction and thus have the same key K_{x+1}, the server is able to decrypt and check the response.

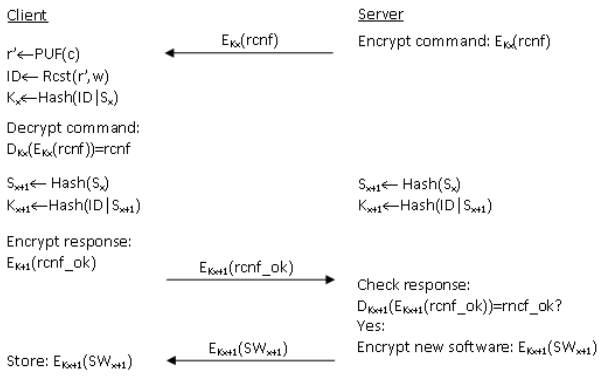

Figure 3: Reconfiguration protocol concept

If the server has validated the response it encrypts the new software SW_{x+1} with the new key K_{x+1} so it can be sent to and run by the client device. If the response is not valid, appropriate measures have to be taken.

5. DESIGN CONSIDERATIONS

5.1 Constraints for secure implementation

With respect to the design of the *Enrolment* protocol, the reason for using a hash function to write the state value into the NVM is that with this construction there is no direct write (or read) path to the memory. This way it will be impossible for an attacker to abuse this channel for writing a chosen value to the state memory to perform some form of illegal reconfiguration. Since the hash function is assumed to be irreversible it is impossible to know which value should be written at the input to get a specific value in the memory.

Note that when using the same hash function for both enrolment (through the external interface) and reconfiguration (input from NVM), it is preferred to change its usage in such a way that an input S_x entered at the external interface results in a different output than when taken from NVM. This is to prevent an attacker being able to input an already known (previously used) S_x into the hash function of the system to get value S_{x+1} back into the memory. This statement holds for all values of x larger than 0, while it is assumed that S_0 is only known to the server and is secure.

The described hash function is required to prevent an attacker from writing a previous state into the NVM. Besides changing this state, it may still be possible for an attacker to perform an illegal re-enrolment: an attacker could be able to set a new value for ID. To prevent an illegal re-enrolment, two possible methods are described that can be implemented in combination with the previously described hash.

The first method is to prevent enrolment physically by disabling the control and data paths associated with enrolment. In practice this means disabling the relevant state transition logic in the control unit and disabling the full width of the external NVM write data path. Anti-fuse technologies based on gate-oxide rupture are available in standard ASIC design flows and are suitable for this type of application.

If using fuses is not an option, it is also possible to prevent illegal re-enrolment using additional security in the proposed protocols. This way re-enrolment by an unauthorized entity can be prevented, while it is still possible for an authorized entity to re-enrol. This is in contrast to using fuses, when

any form of re-enrolment becomes impossible. To demonstrate this method, we assume three possible attacks that can be performed. First, an attacker wants the device to execute a previous enrolment command. This replay-attack can be prevented by the inclusion of a parameter corresponding to the current state in the enrolment command (e.g. encryption with the current key). Preventing an attacker from executing his own enrolment command, authentication can be performed before enrolling (e.g. signing procedure with public/private keys). The final example is an attacker wanting to run a corrupted command. This can be prevented by an integrity check over the data of the command.

Note that methods from this section are suggestions by the authors, other methods might also work. Details of solutions will depend on the specific implementation of an LR-PUF.

5.2 Implementation cost estimate

To estimate the amount of resources required for implementing the memory-based LR-PUF, an architecture design is required as an example. We will use the design from figure 4 for our estimate. The figure omits the control logic for clarity. The design contains the following components:

- **Fuzzy Extractor.** As an example, an error correction code construction from [1] has been used. This code is a concatenation of a repetition [11,1,11] with a Golay [24,12,8] code (used to to reconstruct an ID of 171 bits).
- **0.5kB of SRAM memory**, since the Fuzzy extractor requires 495 bytes of SRAM as its input.
- **0.5kB of public NVM** to store helper data. This size needs to be equal to the SRAM. NVM is assumed be Flash or EEPROM external to the SOC embedding the LR-PUF.
- **256 bits programmable private NVM** for storing the state. NVM technologies which can be implemented in standard CMOS processes have been proposed since at least the early 1990s [12], although it is only relatively recently that they have become widely available in standard IC design flows. These technologies, generically known as "logic NVM", are a good fit for the state storage requirement.
- **Hashing functionality** consists of SHA-256 algorithm.
- **Encryption/Decryption** uses AES-128 in CBC-mode.
- **Control logic** (memory controllers, muxing, I/O, etc.).
- **One-time programmable fuses** are used to disable data and control paths related to enrolment. Anti-fuse technologies based on gate-oxide rupture are available in standard IC design flows at sizes ranging from 16 fuses upwards.

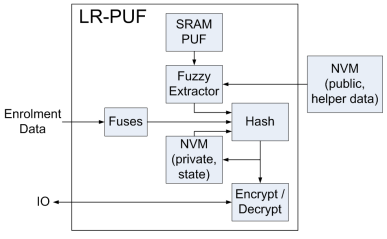

Figure 4: Example of LR-PUF architecture

For each of these components an area estimation in GE [1] has been made for implementation in 65nm technology. These estimations can be found in Table 1.

[1]GE - Gate Equivalent is a measure of area in any technology. 1 GE is the area of a NAND2 (standard drive strength).

Table 1: Area estimate for implementation in 65nm

Component	Estimate	Comments
SRAM and control logic	10k GE	Includes LR-PUF control
Public NVM	-	External Flash/EEPROM
Private NVM	128 GE[2]	256-bit 2T MTP NVM
FE	4k GE	Repetition and Golay
Hashing	20k GE	
En-/Decrypt.	10k GE	128 bits key length
Fuses	32 GE	64 2T OTP anti-fuses
Total	44.2k GE	

Naturally, the total resource cost of the memory-based LR-PUF is higher than the cost of storing a key in NVM or using a regular PUF (which can both be done with a subset of the components from the list above). However, it is clear that neither of these two solutions suffice for the use case that has been stated in this document. Storing a key in NVM is susceptible to both cloning and downgrading (as defined in section 2). Therefore, this method of key storage is not at all comparable to the LR-PUF. Comparing the LR-PUF with a regular PUF implementation (which is still vulnerable to downgrading) shows that the only component that is required for the LR-PUF and not for a regular PUF is the private NVM. Hashing and en-/decryption functionality may be optional in some PUF implementations, but are usually required for secure key storage implementations. Based on this comparison we conclude that the additional protection against downgrading attacks, as provided by the LR-PUF, comes at only little additional resource cost.

6. SECURITY ANALYSIS

In this section we informally analyse the secure key storage solution developed in previous sections in the context of the attacker model from section 3.

6.1 Cloning attack

The first type of attack we consider are cloning attacks. As described in Section 2, a typical cloning attack involves extracting the cryptographic secret from the target device and copying it to the clone. For LR-PUF-based secure key storage a cloning attack of this type would require the LR-PUF state and PUF response to be replicated in the cloned system. The properties of the underlying PUF primitive are sufficient to prevent such simple copying attacks since the PUF is physically unclonable.

More sophisticated cloning attacks can be envisaged which exploit the mathematical cloneability of memory-based PUFs. If the PUF response r can be successfully extracted, then a cloned device might embed a circuit modelling the PUF behaviour, something that is distinct from attempting to physically clone the PUF. A variant of this attack might attempt to extract and replicate the PUF-derived secret ID rather than r. It is assumed in both cases that the attacker has knowledge of the current state S, perhaps also through invasive attack methods.

For any modern deep-submicron technology the barriers to a successful execution of the above attack are considerable. The attacker would need access to high-end probing

[2]Area estimate omits NVM chargepump.

technologies for extracting r, be capable of reverse engineering the SOC embedding the LR-PUF and finally fabricating the functionally equivalent clone. The associated costs are significant with no guarantee of success.

6.2 Downgrading Attack

The second considered attack type is an LR-PUF state modification attack where the attacker attempts to replicate a previous state by writing it to the NVM. We assume that the attacker has knowledge of a previous valid state. Writing arbitrary data to the NVM is prevented due to the presence of a hash function in the NVM write path. However, this mechanism does not prevent an attacker from attempting to write a previously known state S_x in order to write S_{x+1} to the NVM. As described in Section 5 such an attack is prevented at hardware level by disabling the external NVM write path after enrolment, or by using different cryptographic hashes for enrolment and reconfiguration. An attacker must therefore resort to invasive methods. For NVM technologies based on a floating-gate charge storage mechanism, an invasive attacker's options for writing an arbitrary state would be to manipulate the NVM cell array directly or manipulate the NVM write path. The technical challenges and cost of such attacks are considerable, particularly when compared to the outcome - a downgrade attack on a single device instance. It is likely that the attack cost is significantly greater than the potential benefit to the attacker, and can be ruled out.

7. CONCLUSIONS

In this paper we have presented a new type of LR-PUF, which bases its security on a combination of the physical properties of a memory-based PUF and state information stored in NVM. This LR-PUF is specifically suitable for secure key storage in a system that requires the key to be updated. The new construction is shown to be more secure than storing a key in NVM. Furthermore, the fact that the stored key can be updated in the LR-PUF is an additional feature in comparison to a regular PUF. A comparison of the resource costs for an LR-PUF and a regular PUF has shown that the new functionality comes at only little additional resource costs. Finally, a security analysis has shown that it is economically not viable to perform an attack on this LR-PUF, when the design is according to the requirements for secure design from this paper. The authors conclude that this paper introduces a new type of LR-PUF that allows a hardware intrinsic key to be stored and updated in a secure and resource efficient manner.

Acknowledgement

The authors thank Erik van der Sluis and Peter Simons for their help with hardware implementation and all involved partners of the UNIQUE project for their feedback and discussions. This work has been supported by the European Commission through the FP7 programme UNIQUE.

8. REFERENCES

[1] C. Bösch, J. Guajardo, A.-R. Sadeghi, J. Shokrollahi, and P. Tuyls. Efficient helper data key extractor on fpgas. In *Proceedings of CHES*, pages 181–197, 2008.

[2] Y. Dodis, R. Ostrovsky, L. Reyzin, and A. Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. *SIAM J. Comput.*, 38:97–139, March 2008.

[3] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas. Silicon physical random functions. In *Proceedings of the 9th ACM conference CCS'02*, pages 148–160, New York, NY, USA, 2002. ACM.

[4] 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 international workshop on Cryptographic Hardware and Embedded Systems*, CHES '07, pages 63–80, Berlin, Heidelberg, 2007. Springer-Verlag.

[5] S. Katzenbeisser, U. Kocabas, V. van der Leest, A. Sadeghi, G. Schrijen, H. Schröder, and C. Wachsmann. Recyclable pufs: Logically reconfigurable pufs. In *Proceedings of CHES, to appear*, 2011.

[6] S. Kumar, J. Guajardo, R. Maes, G.-J. Schrijen, and P. Tuyls. Extended abstract: The butterfly puf protecting ip on every fpga. In *Proceedings of HOST 2008*, pages 67 –70, june 2008.

[7] K. Kursawe, A.-R. Sadeghi, D. Schellekens, B. Skoric, and P. Tuyls. Reconfigurable physical unclonable functions - enabling technology for tamper-resistant storage. In *Proceedings of HOST 2009*, pages 22–29, Washington, DC, USA, 2009. IEEE Computer Society.

[8] J. Lee, D. Lim, B. Gassend, G. Suh, M. van Dijk, and S. Devadas. A technique to build a secret key in integrated circuits for identification and authentication applications. In *VLSI Circuits, 2004. Digest of Technical Papers. 2004 Symposium on*, pages 176 – 179, june 2004.

[9] D. Lim, J. Lee, B. Gassend, G. Suh, M. van Dijk, and S. Devadas. Extracting secret keys from integrated circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13(10):1200 –1205, oct. 2005.

[10] R. Maes, P. Tuyls, and I. Verbauwhede. Intrinsic pufs from flip-flops on reconfigurable devices. In *Workshop on Information and System Security (WISSec 2008)*, page 17, Eindhoven,NL, 2008.

[11] R. Maes and I. Verbauwhede. Physically unclonable functions: A study on the state of the art and future research directions. In D. Basin, U. Maurer, A.-R. Sadeghi, and D. Naccache, editors, *Towards Hardware-Intrinsic Security*, Information Security and Cryptography, pages 3–37. Springer Berlin Heidelberg, 2010.

[12] K. Ohsaki, N. Asamoto, and S. Takagaki. A single poly eeprom cell structure for use in standard cmos processes. *Solid-State Circuits, IEEE Journal of*, 29(3):311 –316, Mar. 1994.

[13] P. S. Ravikanth. *Physical one-way functions*. PhD thesis, 2001. AAI0803255.

[14] B. Skoric, P. Tuyls, and W. Ophey. Robust key extraction from physical uncloneable functions. In J. Ioannidis, A. Keromytis, and M. Yung, editors, *Applied Cryptography and Network Security*, volume 3531 of *Lecture Notes in Computer Science*, pages 99–135. Springer Berlin / Heidelberg, 2005.