MAC Precomputation with Applications to Secure Memory*

Juan A. Garay[†] Vladimir Kolesnikov[‡] Rae McLellan[‡]

Abstract

We present *ShMAC* (Shallow MAC), a fixed input length message authentication code that performs most of the computation *prior* to the availability of the message. Specifically, ShMAC's message-dependent computation is much faster and smaller in hardware than the evaluation of a pseudorandom permutation (PRP), and can be implemented by a small *shallow* circuit, while its precomputation consists of one PRP evaluation.

A main building block for ShMAC is the notion of *strong differential uniformity* (SDU), which we introduce, and which may be of independent interest. We present an efficient SDU construction built from previously considered differentially uniform functions.

Our motivating application is a system architecture where a hardware-secured processor uses memory controlled by an adversary. We present in technical detail a novel, more efficient approach to encrypting and authenticating memory and discuss the associated trade-offs, while paying special attention to minimizing hardware costs and the reduction of DRAM latency.

Key words: Message authentication code (MAC), MAC precomputation, System on a Chip, tamper-resistant hardware.

1 Introduction

With the highly publicized attacks on consumer computer products, such as the iPhone [14] and Xbox, security of computing has become a topic of widespread commercial interest. Broadly speaking, security of computing can be divided into two main areas — hardware and software security. Software security is concerned with integrity of the software and whether it can be suborned to yield control or reveal sensitive information to an outside attacker. Hardware security, on the other hand, assumes that the adversary has full physical access to the device and may use oscilloscopes and logic analyzers to observe and compromise the computing system. This paper focuses on ways to efficiently provide hardware security. For that purpose, we present a new MAC technique, and discuss its application in securing memory.

Recent VLSI advances have provided strongly tamper-resistant hardware computing platforms by integrating complete Systems on a Chip, through SoC technology. It is considered infeasible to all but government-scale attackers to perform meaningful analysis of the internals of production SoC. Ideally, we would store and execute the entire computation on a SoC, eliminate external DRAM (Dynamic Random Access Memory), and encrypt all off-chip communication. However, this is not possible in most practical

*An abridged version of this paper appears in the proceedings of the *12th Information Security Conference – ISC 2009*.

[†]AT&T Labs – Research, 180 Park Ave., Florham Park, NJ 07932. E-mail: `garay@research.att.com`. Work partly done while the author was at Bell Labs.

[‡]Bell Labs, 600 Mountain Ave., Murray Hill, NJ 07974,USA. E-mail: `{kolesnikov|rae}@research.bell-labs.com`.

scenarios, due to prohibitive costs of such large SoC. In this paper we consider the question of how to encrypt off-chip DRAM transactions with minimal performance degradation and cost increase. Note that such transactions occur much more frequently than network messages and have much more stringent latency requirements. Since processor performance is so tightly dependent on off-chip memory latency, speeding up the encryption/authentication process is of primary importance.

For many on-chip bus protocols (e.g., [1, 17]) the address is available early in the bus transaction between the processor and memory controller, while the larger-size data follows later and is composed of multiple transfers of sub-units. Such serialization of data transfers in on-chip buses is an engineering trade-off between performance and the number of wires required for a wider bus. Therefore, an encryption/authentication algorithm which can postpone data-dependent computation, can start earlier in the memory transaction and potentially reduce the performance impact of an encrypted memory system. This paper describes an efficient way to encrypt off-chip memory transactions and provide data authentication that takes advantage of the early arrival of the memory address.

1.1 Our Contributions

Our main contribution is a new fixed input length Message Authentication Code (MAC) construction which allows the bulk of the MAC computation to be performed *before* the message m is available. The computation dependent on m is the evaluation of (a new variant of) an ϵ*-differentially uniform* (ϵ-DU) function [25, 24] (cf. Section 2) and an XOR operation, which is much simpler and faster than a typical MAC implementation via a block cipher. In envisioned instantiations, MAC precomputation is a PRP (e.g., a full 10-round AES) evaluation, and the remaining computation (dependent on m) is an evaluation of 2- or 4-round AES.

As a second contribution, we present a secure DRAM architecture, discussing at length security/efficiency trade-offs and underlying design choices.

1.2 Related Work

As our work consists of two relatively independent (but complementary) contributions – a cryptographic construction and a secure DRAM design – we separate the discussions of related work accordingly below. First, we discuss Wegman-Carter [31] and related constructions, followed by an overview of previous work on secure DRAM.

On precomputation in Fixed Input Length MACs. Some details and applications of the specific property of MAC precomputation have been discussed in the literature (e.g., [21]), although, to our knowledge, not in the severely restricted environments (with respect to both data-dependent computation and precomputation time and chip surface area) that we consider. In this section we overview previous work on message authentication, with emphasis on precomputation. We discuss the relationships between the building blocks, clarify the terminology and review some efficient constructions.

We are mainly interested in validating 256-bit data blocks. The direct approach is to simply encrypt (e.g., with a blockcipher, such as AES) the data concatenated with the address, and possibly some redundancy. However, this solution is unsatisfactory since it does not allow for precomputation, and, further, requires both encryption and decryption hardware.

Before discussing previous work in more detail, we recall some definitions. Let $H : K \times X \rightarrow Y$ be a function family, indexed by the key $k \in K$. A Universal Hash Function (UHF), or universal$_2$, H guarantees that $\forall x_1 \neq x_2 \in X, \Pr_k[H_k(x_1) = H_k(x_2)] \leq \frac{1}{|Y|}$. That is, no pair of preimages is mapped into the same value by more than one $|Y|$-th of the functions. A stronger notion of Strongly Universal (SU) H requires that $\forall x_1 \neq x_2 \in X, \forall y_1, y_2 \in Y, \Pr_k[H_k(x_1) = y_1 \wedge H_k(x_2) = y_2] = \frac{1}{|Y|^2}$. In other words, H_k maps all distinct x_1, x_2 independently and uniformly.

One of the most celebrated MAC schemes, and also one that naturally allows precomputation, was proposed by Wegman and Carter [31]. Extending the authors' previous work on UHF families, in [31] they introduced the notion of SU hash families, and showed that $MAC_{k,r}(m) = H_k(m) \oplus r$ is an unconditionally secure MAC, where H is an SU function, r is a one-time pad, and k is a random index into the family H.

Stinson [26] formalized the notion of ϵ-Almost SU (ASU), a more general class of functions usable with the Wegman-Carter MAC construction. As the name suggests, ASU functions simply allow less strict bounds on the probability guaranteed by SU. Stinson also showed how to combine a (faster) UHF with an ASU function to obtain a faster ASU function. Brassard [4] pointed out that a pseudorandom generator could be used in place of one-time pad. Krawczyk [20] noticed that ϵ-Almost XOR Universal (AXU) function families, weaker than ASU's, are sufficient for Wegman-Carter MAC. (Recall that H is ϵ-AXU, if $\forall x_1 \neq x_2 \in X, \forall c \in Y, \Pr_k[H_k(x_1) \oplus H_k(x_2) = c] \leq \epsilon$. Krawczyk called this notion *otp-secure*, but AXU is the more frequently used term today.)

Following these fundamental results, a lot of work went into the design of efficient universal, AU, ASU and AXU functions. Most of the research concentrated on software-efficient functions, i.e., those that take advantage of CPU's instruction sets which, in particular, include multiplication. Unfortunately, algebraic solutions are not acceptable in our setting, due to the latency and cost of hardware implementation of multiplication.

In fact, acceptable solutions would only be those that reuse the circuitry of the PRFG to generate the pad r and to evaluate H. Our solution does just that. Alternatively, a MAC scheme with a similar performance can be extracted from a large volume of previous work. Several papers contribute pieces of the total solution, but, to our knowledge, none explicitly states it; further, several sources use conflicting terminology.

Firstly, we point out that neither UHF nor AU functions are sufficient for Wegman-Carter MAC security. This is because they do not guarantee that an offset in the argument will not result in an unpredictable offset in the value of H_k. For example, the identity function is a UHF, but clearly a Wegman-Carter MAC based on it is easily forged. We note, however, that UHF and AU are often used in MACs for efficiency reasons, but only as *part* of the function H; a stronger ASU component is additionally required in H [26]. Further, some sources (e.g., [32]) "blend" the notions of UHF and SU, in fact defining UHF as SU.

Therefore, although it was previously observed, in [24], for example, that it is possible to obtain AU functions from four-round AES, such results are not applicable for our uses of Wegman-Carter MAC. To our knowledge, the only explicit AXU construction from an ϵ-DU function recently appeared in [18]. In particular, it uses AXU derived from a 4-round AES in the Wegman-Carter MAC. That MAC construction, however ([18], Algorithm 1), generates fresh keys for H and the pad r for each MAC evaluation, which is an unacceptable overhead for our setting. We observe, however, that in our setting, the keys of H could be reused, which would bring the resource requirements down to those in our proposed construction.

Related work on secure memory. There is a vast amount of work on securing memory. One direction uses smart cards or other separate adjunct chips such as TPMs (Trusted Platform Module) [29]. These methods are usually limited; for example, they do not protect intellectual property contained in the software running on an (insecure) host, but only secure execution of small parts of it by running it on the smart card/TPM. An interesting use of a smart card processor was proposed in the XμP system [5]. XμP allows the ROM-less smart card to execute signed code, using the terminal as a (cheap) storage. [5] describes ways of securing the computation, including a public key and symmetric key-based authentication of the executing code. At a high level, the symmetric-key case resembles our setting; however, XμP is not as severely restricted, uses computationally expensive hash functions, and does not attempt MAC precomputation.

Another system is XOM [22], which provides architectural support for software licensing and allows code to be authenticated and run even under untrusted operating systems. XOM requires a significant modification of the processor's instruction cache, the addition of special instructions, and operating system (OS) support. Our system is more general, is independent of the instruction set, and supports any processor architecture.

Closer to our setting, securing memory in a SoC system was announced by IBM [15] and considered academically (e.g., AEGIS [28, 27], CryptoPage [9], TEC-Tree [11]). These systems validate memory by maintaining a hash tree of the entire DRAM, each transaction requiring 20-30 DRAM accesses and hash evaluations. Caching part of the tree somewhat reduces the performance impact [13] at the cost of on-chip resources. Solutions to the similar problem of "online memory checking" (see [3, 10] and references therein), where the checker (processor) ensures (only) the integrity of adversarially controlled storage (RAM), also incur a logarithmic overhead. Our MAC is an order of magnitude faster (but with weaker replay protection). We believe such compromise is well suited for many industrial applications.

Other systems such as the one presented in [30], PE-ICE [12] or TEC-Tree [11], forgo Merkle trees but require significant on-chip storage for nonce or checksum values updated on each memory write. While the amount of on-chip storage can be as small as a byte for each encrypted off-chip storage block, this method doesn't scale to to support the desired gigabytes of off-chip DRAM. Further, it can be shown that "natural" CRC (Cyclic Redundancy Code)-based integrity checking mechanisms (e.g., [30]) have critical vulnerabilities (see Section 5).

Given these overheads, we choose to forgo replay attack protection, but instead mitigate the threat by changing the encryption keys at reasonably frequent intervals. In encryption and authentication, we focus on efficiency and minimal additional on-chip resources. In our system, authenticating a memory access takes slightly more than a PRP evaluation, and is effectively further reduced by the precomputation of the MAC.

1.3 Organization of the Paper

In Section 2 we introduce the necessary notation, definitions and building blocks that we will be using. Section 3 is the cryptographic core of this work. We first discuss the intuition behind, and then formally present our MAC construction — *ShMAC*, together with an evaluation of its performance and instantiation considerations. In Section 4 we present the system aspects of our secure memory architecture. In particular, we discuss the assumptions, security objectives, and restrictions of our system, and its use of ShMAC. We conclude the paper with Section 5, where we analyze vulnerabilities of a proposed system attempting to achieve low latency using CRC-based integrity checking.

2 Preliminaries

We denote the security parameter by k, keys by $\ell \in \{0,1\}^k$, a pseudorandom permutation generator by PRPG, and a pseudorandom permutation by PRP. The constructions in this work assume the existence of PRPGs.

2.1 Message Authentication Code (MAC)

A MAC is a tool for ensuring data integrity. It is most commonly used in authenticating communication, and we use it in a similar setting. In our setting, the data is stored in an untrusted location and MAC is used to ensure its integrity.

In a traditional MAC, the tag generation function is stateless and deterministic, and verification is done by applying the tagging function to compute the correct tag of the given message, and comparing it with the candidate tag. We need a slightly more general notion, which we call a *nonce-based MAC*, and which allows the generation function to use nonces. More formally:

Definition 1 *A* nonce-based message authentication code *is a stateless algorithm* $MAC : \{0,1\}^k \times \{0,1\}^k \times \{0,1\}^* \to \mathrm{TAG}$*, which on input key* $\ell \in \{0,1\}^k$*, nonce* $r \in \{0,1\}^k$ *and message* $m \in \{0,1\}^*$*, outputs a tag* $t \in \mathrm{TAG}$*. (Here* TAG *is the domain of tags, which depends on* k*.) We will sometimes write* $MAC_{\ell,r}(m)$ *to mean* $MAC(\ell, r, m)$*; we will also sometimes omit* r *and just write* $MAC_\ell(m)$ *for simplicity.*

Now let $\ell \in_R \{0,1\}^k$*, and* $\mathcal{A}$ *be a nonce-respecting polynomial-time adversary with access to oracle* $\mathcal{O}(r,m) = MAC_{\ell,r}(m)$*.* $\mathcal{A}$ *outputs a message* m' *and its alleged signature (i.e., a nonce-tag pair)* $\tau' = (r', t')$*, subject to the condition that it never received* t' *from* $\mathcal{O}(r', m')$*. We say that* MAC *is* secure *if for every such* $\mathcal{A}$*,* $\Pr[MAC_{\ell,r'}(m') = t'] < 1/k^c$ *for every* c *and sufficiently large* k*.*

In the above definition, by "nonce-respecting adversary" we mean an adversary who never queries the MAC oracle with the same nonce twice. We give $\mathcal{A}$ the freedom to choose his nonces at will with the single above restriction. Throughout the paper, all our adversaries are nonce-respecting.

We remark that, although we define MAC in its commonly encountered general form, in our application we will use the fixed-length variant of this definition, and specifically for messages of length k, i.e., $m \in \{0,1\}^k$ rather than $m \in \{0,1\}^*$. Further, it will be convenient for us to use keys longer than k bits, and thus we allow $\ell \in \{0,1\}^{ck}$, where c is a small constant (e.g., $c = 2$).

We note that Definition 1 imposes a *strong unforgeability* property [2], which enforces that $\mathcal{A}$ cannot create new valid tags on the old (i.e., already tagged) messages. In contrast, "regular" message authentication schemes often do not consider a forgery a valid message-signature pair (m, τ') when the oracle was queried on m and returned $\tau \neq \tau'$. In our application, however, strong unforgeability is essential. We note that, as a side effect, strong unforgeability allows us to avoid the introduction and discussion of verification oracles in the definition of MAC. (See [2] for further discussion on this topic.)

We remark that, in practice, MAC schemes are built directly from PRPGs. Similarly to PRPGs, practical MAC schemes are not defined for all k, but rather, for some fixed but sufficiently large k. Our MAC construction will follow the latter paradigm, but we will perform the analysis in the asymptotic setting.

2.2 ϵ-Differential Uniformity and Properties of AES Rounds

A main building block for our MAC construction is *Strongly Differentially Uniform* (SDU) functions, introduced in Section 3.2. An SDU function family is a stronger version of a *Differentially Uniform* (DU) family, which is widely used in block cipher design and which we now present as background.

For our application we will use the sub-class of *keyed* ϵ-DU *permutations*, due to their efficiency. Therefore, for simplicity, we do not discuss here ϵ-DU functions in their full generality. However, we note that unkeyed permutations or functions [25] could also be used in our constructions, and our analysis (appropriately modified for indices, etc.) equally applies.

Let $\mathcal{G} : \{0,1\}^k \times \{0,1\}^k \to \{0,1\}^k$ be a keyed permutation family. Let $\Delta x, \Delta y \in \{0,1\}^k$ be fixed and let $X \in \{0,1\}^k$ be a uniformly distributed random variable. Let $G_\ell \in \mathcal{G}$. (In the sequel, we may sometimes omit index ℓ and write $G \in \mathcal{G}$, when ℓ is clear from the context or where it does not play a role.) The *differential probability* $DP(\Delta x, \Delta y, \ell)$ is defined as

$$DP(\Delta x, \Delta y, \ell) = \Pr_X[G_\ell(X) \oplus G_\ell(X \oplus \Delta x) = \Delta y]. \tag{1}$$

Here Δx and Δy are viewed as input/output differences. The *expected* differential probability $EDP(\Delta x, \Delta y)$ is the expectation of $DP(\Delta x, \Delta y, \ell)$, over all keys ℓ. We are interested in the *maximum EDP* (*MEDP*):

$$MEDP(\mathcal{G}) = max_{\Delta x, \Delta y \in \{0,1\}^k \setminus 0} EDP(\Delta x, \Delta y). \tag{2}$$

Informally, a small *MEDP* value corresponds to good bit mixing by $\mathcal{G}$ — indeed, small *MEDP* means that any change in the (randomly chosen) input of the cipher results in an unpredictable output. However, small *MEDP* does not necessarily imply "security under multiple queries," since the *MEDP* experiment is defined over all keys ℓ.

Definition 2 *We say that a permutation family $\mathcal{G}$ as defined above is ϵ-*Differentially Uniform *(ϵ-DU), if MEDP($\mathcal{G}$) $\leq \epsilon$.*

It is well known [19, 8] that the *MEDP* of two-round AES (AES2) is at most $1.6 \cdot 2^{-28}$, and the *MEDP* of four-round AES (AES4) is at most $1.8 \cdot 2^{-110}$. Thus, AES2 is a $1.6 \cdot 2^{-28}$-DU permutation, and AES4 is a $1.8 \cdot 2^{-110}$-DU permutation.

3 *ShMAC*: MAC with Precomputation

In this section we present *Shallow MAC* (ShMAC), a MAC scheme which takes advantage of precomputation. The required precomputation essentially consists of one PRP evaluation, while the message-dependent portion is a small shallow circuit, which can be evaluated in a fraction of the time required for a PRP evaluation. (As a bonus, in our envisioned instantiation, precomputation can share hardware gates with the rest of MAC computation. This is a critical advantage in cases where chip area is restricted, as it is in FPGAs.)

Recall that we require a low-latency MAC scheme, simultaneously "cheap" to implement in hardware, and faster than the evaluation of a PRP (e.g., AES) or a hash function. This requirement precludes many standard MAC solutions, such as AES-based, which require availability of the message at the beginning of the computation. (To be concrete about the involved latencies, recall that AES requires the sequential evaluation of at least 10 rounds[1]. Further, many (but not all) Universal Hash Function (UHF)-based constructions require expensive group arithmetic and additional hardware, and thus are unacceptable in this setting. See Section 1.2 for more details.)

However, as also discussed in Section 1, in many systems the address of the memory transaction arrives before the data, and thus the hardware MAC unit is idle waiting for the data. We explore the possibility of using these idle cycles to perform *precomputation* to speed up MAC generation.

3.1 The Intuition behind ShMAC

An ϵ-DU permutation family $\mathcal{G}$ (e.g., 2-round AES), an object with much weaker security properties than a PRPG, can in principle be the core of a MAC, with appropriate pre- and post-computation. Indeed, $G \in_R \mathcal{G}$ provides good bit mixing, but only on random inputs. We satisfy this by using (nonce-based new and secret) precomputed randomness to mask the data d prior to each application of G. This masking of the inputs additionally prevents adversary $\mathcal{A}$ from collecting any information on (the key of) G, even if $\mathcal{A}$ sees MAC evaluated on messages of his choice (i.e., queries the MAC oracle adaptively).

Note that even though $\mathcal{A}$ has no knowledge of the random mask $mask_r$ derived from nonce r, he can attempt a forgery using the same r (and thus the same $mask_r$). The output unpredictability guarantees of DU functions are too weak to protect against this attack, since in our scenario $\mathcal{A}$ knows $G(d \oplus mask_r)$[2]. We strengthen the notion of DU to preserve its guarantees even after one query to G – see Definition 3 below. In terms of implementation, it turns out that masking the output of G with fixed secret randomness (which can be viewed as part of G's key) is sufficient to satisfy the stronger requirements, and results in a secure MAC.

[1]For our application, fewer rounds (e.g., 8) would provide an adequate level of security, because the keys are refreshed frequently, and $\mathcal{A}$ would only have on the order of seconds or minutes to "crack" the MAC.

[2]It is easy to see that if G is unkeyed, $\mathcal{A}$ can easily construct a forgery.

3.2 ϵ-Strongly Differentially Uniform Functions

In this section we introduce the notion of *Strong Differential Uniformity* (SDU), discuss its relationship with DU, and present an efficient construction. The new notion is a natural building block in MAC constructions, including ours, and may have applications in other areas. For simplicity, we give an asymptotic notion of ϵ-Strongly Differentially Uniform (ϵ-SDU) permutations, by allowing ϵ to be a function of the security parameter k.

Definition 3 *Let $\mathcal{G} : \{0,1\}^k \times \{0,1\}^k \to \{0,1\}^k$ be a permutation family indexed by security parameter k, and $\mathcal{A}$ be a computationally unbounded TM. Consider the following experiment $\mathsf{SDU}_{\mathcal{A},\mathcal{G}}(k)$:*

1. *$G \leftarrow \mathcal{G}$ is selected at random by choosing the key. Further, a random $R \in \{0,1\}^k$ is chosen.*
2. *$\mathcal{A}$ provides d, and receives $G(d \oplus R)$. $\mathcal{A}$ outputs $\Delta x, \Delta y \in \{0,1\}^k \setminus 0$.*
3. *The output of the experiment is defined to be 1 if $G(d \oplus R) \oplus \Delta y = G(d \oplus R \oplus \Delta x)$, and 0 otherwise.*

We say that $\mathcal{G}$ is $\epsilon(k)$-Strongly Differentially Uniform (ϵ-SDU for short), if for all $\mathcal{A}$, $\Pr[\mathsf{SDU}_{\mathcal{A},\mathcal{G}}(k) = 1] \leq \epsilon(k)$, where the probability is taken over the random choices used in the experiment.

It is easy to see how the ϵ-SDU notion is derived from ϵ-DU's. Indeed, Definition 2 can be cast asymptotically and in game style, resulting in exactly Definition 3, with the exception that in the corresponding experiment $\mathsf{DU}_{\mathcal{A},\mathcal{G}}(k)$, $\mathcal{A}$ is not given $G(d \oplus R)$. Note that the notion of ϵ-SDU is strictly stronger than that of ϵ-DU. Indeed, while unkeyed ϵ-DU functions exist [25], unkeyed ϵ-SDU functions don't. (This is because $\forall \Delta x$, an ϵ-SDU $\mathcal{A}$ can output a winning Δy since he can invert the received $G(d \oplus R)$.)

We now show how to construct an efficient ϵ-SDU permutation from any ϵ-DU permutation, such as AES, at additional negligible cost.

Lemma 1 *Let k be a security parameter, and $\mathcal{G}'$ be a keyed (or unkeyed) ϵ-DU permutation family. Let $\mathcal{G} = \{G = G' \oplus \ell_1 | G' \in \mathcal{G}', \ell_1 \in \{0,1\}^k\}$ be a family additionally keyed by uniformly chosen $\ell_1 \in_R \{0,1\}^k$. Then $\mathcal{G}$ is an ϵ-SDU permutation family, for the same ϵ.*

Proof. First, clearly, $\mathcal{G}$ is a permutation family iff $\mathcal{G}'$ is a permutation. We now prove security properties. Suppose $\mathcal{G}$ is not ϵ-SDU. We show that then $\mathcal{G}'$ is not ϵ-DU. Let $\mathsf{SDU}_{\mathcal{A},G=G'\oplus\ell_1,R,\ell_1}(k)$ be an instance of ϵ-SDU experiment, and $\mathsf{DU}_{\mathcal{A}',G',R}(k)$ be its corresponding ϵ-DU instance. Let $\mathcal{A}$ be an ϵ-SDU adversary. We construct $\mathcal{A}$' who, whenever $\mathcal{A}$ wins in $\mathsf{SDU}_{\mathcal{A},\mathcal{G}}(k)$, wins a corresponding ϵ-DU instance. Clearly, such $\mathcal{A}$' wins at least as often as $\mathcal{A}$ (i.e. with prob. at least $\epsilon(k)$), and thus $\mathcal{G}'$ is not ϵ-DU.

$\mathcal{A}$' begins by running $\mathcal{A}$ and receiving d. $\mathcal{A}$' chooses $r' \in_R \{0,1\}^k$ and gives r' to $\mathcal{A}$. $\mathcal{A}$ outputs $(\Delta x, \Delta y)$. $\mathcal{A}$' outputs $d, \Delta x, \Delta y$.

Since $G_{\ell_1} = G' \oplus \ell_1$, if $\mathcal{A}$ wins, $G'(d \oplus R) \oplus \ell_1 = G'(d \oplus R \oplus \Delta x) \oplus \ell_1 \oplus \Delta y$. Then, by canceling ℓ_1, it is easy to see that $d, \Delta x, \Delta y$ output by $\mathcal{A}$' satisfy the DP condition of $\mathsf{DU}_{\mathcal{A}',G',R}(k)$, and thus $\mathcal{A}$' wins. ∎

3.3 ShMAC Construction

Let d be a data block, and $r \in \{0,1\}^k$ be a nonce; in practice r may be a counter or chosen randomly for each MAC evaluation. Let $\mathcal{G}$ be a ϵ-SDU permutation family (Definition 3), where $\epsilon = \epsilon(k)$ is negligible in k. Let G be a random member of $\mathcal{G}$, selected by randomly choosing the key ℓ. Let $F : \{0,1\}^k \to \{0,1\}^k$ be chosen at random, and unknown to the adversary; in practice F is implemented by a PRPG, such as AES.

Construction 1 *Let F, G, r, d be as above. Shallow MAC is the algorithm:*

$$\text{ShMAC}_\ell(r, d) = (r, G(d \oplus F(r))) \tag{3}$$

Theorem 1 *Construction 1 is a nonce-based MAC as defined in Definition 1.*

In Construction 1, we use a PRPG as a source of indexed secret fresh randomness for each evaluation of MAC. For simplicity of the reductions in the proof of our main theorem, we use a slightly different formulation of ϵ-SDU. Instead of R, we offset d with a randomly chosen secret function F, evaluated at a point i chosen by $\mathcal{A}$. It is easy to see that this modification is purely cosmetic, and does not affect the notion of ϵ-SDU. For clarity, we include it here.

Definition 4 *Let $\mathcal{G} : \{0,1\}^k \times \{0,1\}^k \to \{0,1\}^k$ be a permutation family indexed by security parameter k, and $\mathcal{A}$ be a computationally unbounded TM. Consider the following experiment $\mathsf{SDU}_{\mathcal{A},\mathcal{G}}(k)$:*

1. *$G \leftarrow \mathcal{G}$ is selected at random by choosing the key. Further, a permutation $F : \{0,1\}^k \to \{0,1\}^k$ is chosen at random from the domain of all permutations with the same domain and range.*
2. *$\mathcal{A}$ provides a pair (i, d), and receives $G(d \oplus F(i))$. $\mathcal{A}$ outputs a pair $\Delta x, \Delta y \in \{0,1\}^k \setminus 0$.*
3. *The output of the experiment is defined to be 1 if $G(d \oplus F(i)) \oplus \Delta y = G(d \oplus F(i) \oplus \Delta x)$, and 0 otherwise.*

We say that $\mathcal{G}$ is $\epsilon(k)$-Strongly Differentially Uniform (ϵ-SDU for short), if for all $\mathcal{A}$, $\Pr[\mathsf{SDU}_{\mathcal{A},\mathcal{G}}(k) = 1] \leq \epsilon(k)$, where the probability is taken over the random choices used in the experiment.

Proof. We now prove Theorem 1 by showing that the existence of a successful forger for ShMAC implies the existence of an adversary that wins the SDU game of Definition 4 with non-negligible probability[3]. Let $\mathcal{A}$ be such a forger, and $d', \tau' = (r', t')$ the forgery produced by $\mathcal{A}$.

First, suppose nonce r' was never given to the MAC oracle $\mathcal{O}(r, d) = \text{ShMAC}(r, d)$. Then, any polynomial-time $\mathcal{A}$ has only negligible probability of forging the MAC. Indeed, since F is a random permutation, $F(r')$ is random to $\mathcal{A}$. As $t' = G(d \oplus F(r'))$ would be the value of a permutation of a random point, it must also look random to $\mathcal{A}$, and thus $\mathcal{A}$ cannot compute it other than with negligible probability.

Now suppose otherwise, i.e., r' is a nonce previously (once) given to the oracle $\mathcal{O}$. That is, $\mathcal{A}$ adaptively queried $\mathcal{O}(r_i, d_i)$ a number of times, and, in particular, obtained $t = G(d \oplus F(r'))$ from call $\mathcal{O}(r', d)$. Then, $\mathcal{A}$ outputs a valid forgery $d', \tau' = (r', t')$, where either $d' \neq d$ or $t' \neq t$, or both. Suppose further that $\mathcal{A}$ succeeds in producing this forgery with non-negligible probability. We construct an adversary $\mathcal{A}$' who uses $\mathcal{A}$ to win the SDU game of Definition 4, with the same security parameter k.

The construction goes as follows. Following Step 1 of the SDU game, functions F and G are initialized, but kept secret from $\mathcal{A}$' (and $\mathcal{A}$). $\mathcal{A}$' now proceeds as follows. $\mathcal{A}$' initializes $\mathcal{A}$. Let q be the number of queries to the oracle that $\mathcal{A}$ makes. $\mathcal{A}$' chooses $j \leq q$ as the index for his attack. For all r_i, d_i queries from $\mathcal{A}$ with $i \neq j$, $\mathcal{A}$' answers with a random element from the range of G. For the j-th query, however, $\mathcal{A}$' makes his only query (d_j, r_j) to the SDU game oracle; he obtains $t_j = G(d_j \oplus F(r_j))$ from the oracle and forwards t_j to $\mathcal{A}$. Finally, when $\mathcal{A}$ produces the forgery, if the forgery is produced on r_i, $i \neq j$, then $\mathcal{A}$' gives up. However, if the forgery is on r_j, call it $d', \tau' = (r_j, t')$, then $\mathcal{A}$' outputs $d' \oplus d_j, t' \oplus t_j$. This concludes the description of $\mathcal{A}$'. We analyze the probability of $\mathcal{A}$' winning the SDU experiment — i.e. of the output of the experiment being 1.

First, note that $\Pr[i = j] = 1/q$, since $\mathcal{A}$' provides answers which are distributed statistically close to the oracle answers of the MAC game, which is what $\mathcal{A}$ expects to see. (This is because G is a permutation, and F is random. Thus, the output of the oracle is that of the SDU oracle $G(d' \oplus F(r_i))$, and is a uniformly distributed element in the range of G.) We now show that the pair $(\Delta x, \Delta y) = (d' \oplus d_j, t' \oplus t_j)$ that $\mathcal{A}$' outputs satisfies Equation 1 (the differential probability equation) non-negligibly often, which allows $\mathcal{A}$' to win the game. Indeed, since $d', \tau' = (r_j, t')$ is a valid MAC forgery, it must be that $t' = G(d' \oplus F(r_j))$. However, $t_j = G(d_j \oplus F(r_j))$. Therefore, $\Delta y = t' \oplus t_j = G(d_j \oplus F(r_j)) \oplus G((d_j \oplus F(r_j)) \oplus d' \oplus d_j)$.

[3]Note that the adversary in Definition 4 is in fact allowed unbounded computational power; this fact, however, will not be needed in the current reduction.

Thus, $\mathcal{A}$' wins the SDU game non-negligibly often, which contradicts the assumption that $\mathcal{G}$ is a ϵ-SDU permutation family, with ϵ negligible in k. ∎

Note that ShMAC can be executed on multiple data blocks by simple concatenation of MACs of individual blocks. This observation is motivated by the fact that efficient ϵ-SDU functions may not be readily available from the literature for larger data blocks. For simplicity, we state the following lemma for the case of two blocks; it can be naturally extended to any number of blocks.

Lemma 2 *Let F, G, r be as above, and let d_0 and d_1 be data blocks. Then*

$$MAC(d_0, d_1) = (r, G(d_0 \oplus F(r, 0)), G(d_1 \oplus F(r, 1))) \tag{4}$$

is a nonce-based MAC, as defined in Definition 1.

Proof. We show that Equation 4 is a secure double-input-length MAC. First, note that by Theorem 1, each $G(d_i \oplus F(r \parallel i))$, $i \in \{0,1\}$, is a secure unit-input-length MAC, since $(r \parallel i)$ is a nonce in $\{0,1\}^k$ if and only if r is a nonce in $\{0,1\}^{k-1}$. Suppose now that there exists an adversary $\mathcal{A}$ who is able to forge the double-input-length MAC with non-negligible probability. We construct a nonce-respecting $\mathcal{A}$' who forges one of its component MACs.

$\mathcal{A}$' initializes $\mathcal{A}$ and interacts with it as follows. (For its forgery, $\mathcal{A}$ queries the double-input-length MAC oracle $\mathcal{O}_2(r, d) = \text{ShMAC}_\ell(r, d_0 \parallel d_1)$ of Equation 4 for $d_0, d_1 \in \{0,1\}^k$ and $r \in \{0,1\}^{k-1}$.) Upon each query by $\mathcal{A}$ to $\mathcal{O}_2$, $\mathcal{A}$' queries (and obtains) $t_0 = \mathcal{O}(r \parallel 0, d_0)$, $t_1 = \mathcal{O}(r \parallel 1, d_1)$, where $\mathcal{O}$ is the unit-input-length MAC oracle. $\mathcal{A}$' then returns (t_0, t_1) to $\mathcal{A}$.

Consider the case when $\mathcal{A}$ outputs a forgery $(d'_0 \parallel d'_1, r', (t'_0, t'_1))$. It must be that $\mathcal{A}$ never obtained $(t'_0, t'_1) = \mathcal{O}_2(r', d'_0 \parallel d'_1)$. Further, because $\mathcal{A}$ is nonce-respecting, he could have previously made only a single query on nonce r': $(t_{q_0}, t_{q_1}) = \mathcal{O}_2(r', q_0 \parallel q_1)$.

Now, if $q_0 = d'_0$ and $q_1 = d'_1$, then it must be that either $t_{q_0} \neq t'_0$ or $t_{q_1} \neq t'_1$. If $t_{q_i} \neq t'_i$, then $(d'_i, r' \parallel i, t'_i)$ is a valid forgery, which $\mathcal{A}$' outputs. Otherwise, if for $i \in \{0,1\}$, $q_i \neq d'_i$, or if $\mathcal{A}$ did not query $\mathcal{O}_2$ with a nonce r', then $(d'_i, r' \parallel i, t'_i)$ is a valid forgery, which $\mathcal{A}$' outputs. This completes the proof of the lemma. ∎

3.4 ShMAC Instantiation Considerations

Theorem 1 is stated with respect to an ideal object — a randomly chosen function. In practice, this is implemented by means of a PRPG, and therefore Theorem 1 becomes conditional on the existence of PRPGs. Of course, this transition into the computational model improves the chances of $\mathcal{A}$ to forge the MAC, but it can be easily shown that this improvement is negligible. Note that ϵ-DU functions (and thus ϵ-SDU functions) are known to exist, and their use does not constitute an assumption.

As noted previously, "shortcut" 2- or 4-round versions of AES are ϵ-DU permutations. Further, *AddRoundKey*, the final phase of each AES round, implements the transformation of Lemma 1. At the same time, in many hardware implementations, the AES key schedule is precomputed, with round keys being randomly chosen. Such shortcut AES implementations satisfy the stronger ϵ-SDU requirements and are sufficient for security of MAC. In our implementation, we follow this approach.

Depending on the application, the desired input length of MAC may vary. In our encrypted memory system, for example, we operate on 256-bit blocks. We wish to point out several observations that apply to such usage scenarios. First, it is not necessary to use a "wider" (e.g., 256-bit) block cipher as the PRPG F. Wider block ciphers are more expensive, since they aim to achieve strong bit mixing on the full block. For example, 256-bit Rijndael requires 14 rounds, vs. the 10 rounds of its 128-bit AES sibling. In our application F is only a source of randomness, and it is sufficient to execute AES twice with corresponding adjustment of the nonce r to $(r, 0)$ and $(r, 1)$. Second, $\mathcal{G}$ must be chosen properly as well. Similarly to

AES, 256-bit Rijndael achieves good bit mixing after only 4 rounds [7, 6]. Alternatively, we could apply Lemma 2 and execute 128-bit $\mathcal{G}$ (e.g., AES2 or AES4) on each of the two 128-bit halves of masked data.

In our secure memory application, we choose the nonce r to be a concatenation of the address of the memory location and a global RAM transaction counter (the latter may have to be wrapped around for efficiency). This provides a simple and efficient way of generating nonces. Further, this method allows binding the memory value to the memory location, preventing replay of valid data at wrong locations[4]. Another advantage of this nonce choice is that the bulk of the nonce, the memory address, need not be written to memory, as it is managed by underlying subsystems. Further, this method ensures that the nonce is available before the data arrives, thus allowing precomputation.

4 Applications: Secure DRAM

We now give an overview of a SoC-based secure system which makes use of ShMAC. While we are mainly interested in integrity checking, for completeness we also discuss a (weak) form of memory encryption. As noted in Section 1, all system operations (with the exception of memory transactions) take place inside the presumably secure and tamper-resistant chip. Therefore, securing the memory, which might be adversarially controlled, closes the main avenue of attack. We now discuss the hardware aspects of an encrypted memory implementation using ShMAC for authentication, associated trade-offs and improvements; we view this technical discussion as an additional contribution of this paper.

In our discussion, we omit some of the aspects of the system, such as secure-boot procedures, the design of which is not related to MAC. We start by presenting the Encryption/Authentication Unit (EAU), its on-chip location, connectivity and relationship with other units.

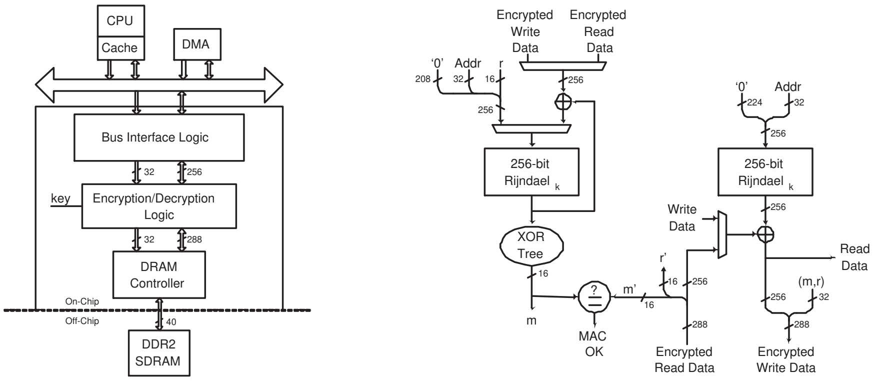
4.1 Overview of Memory Encryption and Authentication

As shown in the conceptual block diagram, Figure 1(a), the EAU is interposed between a conventional DRAM controller and the interface logic that allows potential bus masters, such as CPUs and DMA engines, to access secured off-chip memory. DRAM write transactions are encrypted on the way out to DRAM and read transactions are decrypted coming back from DRAM to the SoC. During the encryption process, a MAC is generated and stored with each encrypted block of memory. During subsequent DRAM read operations, the stored MAC is compared with a newly recomputed MAC to detect corrupted off-chip memory contents.

Each MAC is associated with a fixed number of data bytes, called an *encryption block*, which is the minimal unit of data. That is, the EAU supports only block-size read or write DRAM transactions (and transparently handles creation and verification of the associated MACs). The bus interface logic handles transactions of all sizes. If a bus write transaction affects only a portion of an encryption block, the EAU first needs to read, decrypt and verify the unavailable bits (if any) of the encryption block from off-chip DRAM. Then, it merges the bits to create a full updated encryption block, before it is re-encrypted and written to DRAM.

Encryption block size and the number of bits in the MAC is a complex engineering trade-off. Clearly, each bit of MAC stored in DRAM is unavailable for user data and therefore represents overhead in an encrypting memory system. Short MAC may not not offer sufficient protection against forgers. Since MACs are stored in the same DRAM as the encryption blocks, there is also the physical and costs constraints of the DRAM data width. For most SoC systems, the DRAM is usually 16 or 32-bits wide. Therefore, MACs with that granularity are preferred.

[4] The binding between the data and the nonce is guaranteed by the strong definition of MAC that we use. Indeed, it disallows a poly-time $\mathcal{A}$ to generate new nonce-tag pairs even on previously signed data. Note that this does not prevent replay at the same memory location. We discuss this trade-off in Section 4.



(a) SoC system layout

(b) Conceptual EAU design

Figure 1: SoC-based Encrypted DRAM

Similarly, the size of the encryption block is determined by the range of data sizes expected in typical SoC bus transactions. DMA transfers generate bus transactions of size from single bytes to multi-word IP packets, but CPUs present a characteristic bus transaction width that corresponds to their cache line size. Choosing an encryption block size the same as the cache line will efficiently support the most frequent bus transactions.

Our encrypted memory supports a physical 32-bit wide DRAM system. Encryption blocks are 256-bit wide and the associated MAC can be as short as 32 bits, while providing reasonable security. Each DRAM transaction is therefore eight 32-bit words of data followed by one or two words of MAC. This way, the memory overhead is as low as 12.5% and up to 8/9-ths of the DRAM contents is available for user storage.

Stateless vs. stateful integrity checks. In our design, the EAU is stateless. This is necessary due to severe on-chip resource restrictions. It is not hard to see that encryption and authentication process as described above exposes the system to replay attacks. For example, an adversary can replace the current contents of memory with a value that was stored in that same location previously. Similarly, an adversary can simply not update the DRAM as required by a write transaction. It is easy to see that the system will decrypt and mistakenly accept such data as valid.

Stateful operation is *necessary* to prevent such attacks. Keeping on-chip state per each memory location, however, is prohibitively expensive. A natural solution is to build a Merkle tree [23] of MACs for the entire memory space, as proposed and deployed in, e.g., [16, 28, 27]. However, even with the possible optimizations, maintaining such a tree of MAC values is a performance bottleneck (20-30 memory accesses and hash evaluations for each DRAM transaction) and requires significant on-chip resources, which is unacceptable in many settings, including ours. Instead of expensive tree-based integrity checking, we use a much faster method to achieve a level of security sufficient for most commercial applications.

To limit the exposure to replay attacks, we propose periodic refreshing of encryption keys so as to invalidate sufficiently stale encrypted memory state. It is easy to implement, e.g., by maintaining two memory regions, each encrypted with its own key, and growing one region at the expense of the other. If the keys are refreshed often enough, say, every two minutes, then the window of vulnerability to replay attacks is fairly narrow.

This idle-time key refreshment is much more efficient than maintaining a Merkle tree. We believe that frequent key expiration, and a single MAC per encryption block affords practical levels of security with much less mechanism and performance penalty, and thus is a better security/performance trade-off, suitable for most industrial applications.

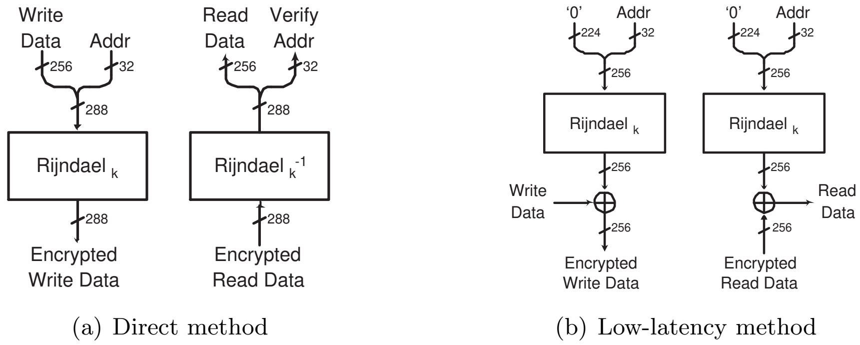


Figure 2: Encryption/authentication methods for off-chip transactions

4.2 EAU Implementation Using ShMAC

The most direct method to encrypt and authenticate off-chip memory transactions, would be to encrypt the concatenated address and data[5]. This would produce a 288-bit encrypted memory write value, as shown in Figure 2(a). However, this method serializes the encryption process with memory write operations and, more importantly, adds decryption delay to the already performance-limiting DRAM read latency. Additionally, this scheme requires both encrypt logic and decrypt logic, which is unacceptable for FPGA implementations.

Motivated by low-latency and small footprint requirements, we prefer a different encryption approach, shown on Figure 2(b), and separate MAC generation from the data encryption process. Figure 1(b) illustrates a conceptual EAU design, described below[6]. A 256-bit pad is generated by Rijndael encryption of the address[7]. The pad is then XORed with the Write data to produce the encrypted result. Since XOR is its own inverse, the same encryption function can be used for both encryption as well as decryption. While encryption remains serialized with the DRAM write operation, the pad calculation can start as soon as the address is available early in the bus transaction. More importantly, for performance-critical read operations, the pad calculation can occur in parallel with DRAM read latency. Once encrypted DRAM data is available, a single XOR operation is the only additional delay incurred by decryption. As a result, decrypted data is returned to the processor with negligible delay[8].

MAC generation proceeds as follows. The PRP F of Construction 1 is achieved by running full Rijndael on the address concatenated with a nonce r. The nonce value can be a global counter that increments with each memory write transaction. We stress that there is no need for expensive pseudo-random generation of the nonce. Note that this first step of the MAC calculation can start as soon as an address is available, simultaneously with the encryption process. The Rijndael output is then XORed with the encrypted data and the same Rijndael data path is reused to compute G of ϵ-SDU family $\mathcal{G}$. In our implementation, G is a four-round Rijndael evaluation[9]. The output of G is collapsed via an XOR tree to a value m, which is concatenated with the original unmodified nonce r to form the MAC written to DRAM — this is the ShMAC output.

In contrast with the decryption process, the MAC verification for memory read operations must first

[5] Additional redundant data can be added under the encryption, if stronger integrity checks are desired.

[6] Reasonable security parameter sizes were included in Figure 1(b) for concreteness, however, their values should be evaluated for specific instantiations.

[7] We note that the 256-bit pad can be more efficiently generated by two parallel 128-bit AES encryptions in fewer rounds. We omit this, as well as other natural optimizations, for the sake of clarity.

[8] Admittedly, reusing the pad for the same DRAM location results in a weakness of the encryption process. However, varying the pad, for example, based on a counter, would preclude pad precomputation for read transactions, or require significant on-chip storage.

[9] As discussed in Section 3.4, we alternatively could use parallel execution of two instances of 2- or 4-round AES.

wait for the DRAM latency in order to acquire the original nonce r, which is stored off-chip. Once data and MAC arrive, F is computed on the address appended with r (14 rounds of 256-bit Rijndael). This value is then XORed with the encrypted read data and the same Rijndael data path is reused to compute G, which consists of four rounds of Rijndael. The XOR tree collapses the result to generate m, which is compared with m', the value of the just-read, off-chip MAC. If they match, the memory read operation is considered uncorrupted.

Note that MAC verification can only start after the original MAC value is read and much later than the actual decryption process, which means that data would have already been returned to the processor before the MAC is verified. We can afford this delay because in our application we consider MAC failure to be so dire that the system effectively resets and discards any use of the corrupted data. Thus, we do not need to implement any recovery mechanisms, such as rollbacks.

Trade-offs and design choices. Due to the unacceptable cost of tree-based integrity checking, it was our decision to use weaker but much more efficient authentication, which allows replay attacks within a small window (e.g., one to several minutes). We believe this is a reasonable compromise. Next, we argue that our authentication approach effectively limits the forger to replay attacks.

Performance considerations require use of short MACs. We first argue that even 16-bit security is sufficient in many practical security applications[10]. (Of course, this parameter would need to be evaluated for each concrete system instantiation, using the following discussion as a guideline.) Indeed, on average, it would take the adversary 2^{15} attempts to forge just one memory block. Note that in our system each unsuccessful attempt would be followed by a forced reboot (a natural reaction to a break-in attempt), which might take around a minute to complete. This means that forging a single block would take an expected 20 days of continuous attacks; forging even two blocks (expected 2^{30} attempts) is infeasible. Thus, attackers are likely to use other attack avenues, such as exploiting the replay permissiveness.

Achieving 16-bit security requires the use of MACs of greater length, since the ShMAC output includes a nonce. In our system, the ShMAC nonce consists of the concatenation of the address and r. We first observe that nonces for different memory locations would never collide; however, nonces may collide within the same memory locations. If many collisions occur, the adversary may eventually accumulate some useful information about G. We mitigate this threat with periodically refreshing F and G (by changing their keys). As an additional disadvantage to the adversary, he does not learn the full value of G's, but only a fraction of it. Thus, we believe that a choice of length for r in the 16–48 bits range would be appropriate for most applications.

Finally, we note the following trade-off regarding the encryption approach. Since we encrypt by XORing data with the pad derived from the memory address, the adversary is able to track changes in data stored in memory. Specifically, when two data blocks d_1, d_2 are encrypted with the same key and stored in the same memory location, the adversary is able to compute $d_1 \oplus d_2$ as the XOR of their encryptions. However, natural attempts to vary the pad, e.g. based on a counter, would either preclude pad precomputation for read transactions, or require significant on-chip storage. We believe that in many commercial applications, this weakness, mitigated in particular by frequent key refreshes, does not significantly help reverse engineering and other hostile analysis and interference.

5 Vulnerabilities of a CRC-based Memory System

As in our case, Vaslin *et al.* [30] aim to achieve low-latency hardware integrity checking. Their idea is to use a highly efficient CRC (CRC8 or CRC32) as the MAC of memory block m, and to encrypt stored data by XORing it with a one-time pad (OTP). For increased security, they propose to store CRC(m)

[10] Of course, by 16-bit security we mean that the probability of a polynomial-time adversary forging a MAC is $(\frac{1}{2})^{16}$, and not that it takes 2^{16} operations to break it.

on-chip, where the attacker cannot access it. It is claimed that "if the data is changed following storage, a CRC of the retrieved value will differ from the stored value."

We observe that in fact, an attacker $\mathcal{A}$ can easily forge the data, and in fact has a lot of flexibility in doing so. Firstly, in many cases $\mathcal{A}$ would *a priori* know the plaintext m of the encrypted data stored in DRAM. (This is because the memory location can store a counter, known code, a network message received from $\mathcal{A}$, etc.) Further, it is trivial to find many $m' \neq m$, such that $CRC(m) = CRC(m')$. Then, $\mathcal{A}$ forges the encryption $m' \oplus otp$ by computing $(m \oplus m') \oplus (m \oplus otp)$. Clearly, this results in the SoC accepting m' as valid.

Furthermore, it is possible for $\mathcal{A}$ to succeed even without having *any* information about m. Indeed, each bit of CRC(m) is an XOR of certain bits of m. The OTP encryption simply offsets the bits of m, but does not "mangle" them. Thus, a flipping of an encrypted bit in DRAM effects a specific (and known to $\mathcal{A}$) sequence of bit flips in the value of $CRC(m)$. Flipping DRAM bits appropriately, $\mathcal{A}$ can easily arrive at a forged value with the same CRC as stored on-chip.

References

[1] ARM. ARM advanced microcontroller bus architecture rev 2.0, 1999.

[2] Mihir Bellare, Oded Goldreich, and Anton Mityagin. The power of verification queries in message authentication and authenticated encryption. Cryptology ePrint Archive, Report 2004/309, 2004. `http://eprint.iacr.org/`.

[3] Manuel Blum, Will Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. In *FOCS '91: Proceedings of the 32nd annual Symposium on Foundations of computer science*, pages 90–99, Washington, DC, USA, 1991. IEEE Computer Society.

[4] Gilles Brassard. On computationally secure authentication tags requiring short secret shared keys. In *Advances in Cryptology – CRYPTO 82*, pages 79–86. Springer-Verlag, 1983.

[5] Benoit Chevallier-Mames, David Naccache, Pascal Paillier, and David Pointcheval. How to disembed a program? Cryptology ePrint Archive, Report 2004/138, 2004. `http://eprint.iacr.org/`.

[6] Joan Daemen. Annex to AES proposal Rijndael. chapter 5. propagation and correlation. Available at `http://www.iaik.tugraz.at/Research/krypto/AES/old/~rijmen/rijndael/PropCorr.PDF`.

[7] Joan Daemen and Vincent Rijmen. AES proposal: Rijndael. Available at `http://www.iaik.tugraz.at/Research/krypto/AES/old/~rijmen/rijndael/rijndaeldocV2.zip`.

[8] Joan Daemen and Vincent Rijmen. Understanding two-round differentials in AES. In *SCN*, pages 78–94, 2006.

[9] Guillaume Duc. Cryptopage. Master's thesis, ENST, Bretagne, June 2004.

[10] Cynthia Dwork, Moni Naor, Guy N. Rothblum, and Vinod Vaikuntanathan. How efficient can memory checking be? In *Theory of Cryptography, TCC 2009*, volume 5444, pages 503–520, 2009.

[11] Reouven Elbaz, David Champagne, Ruby B. Lee, Lionel Torres, Gilles Sassatelli, and Pierre Guillemin. Tectree: A low-cost, parallelizable tree for efficient defense against memory replay attacks. In P. Paillier and I. Verbauwhede, editors, *CHES 2007*, pages 289–302, Berlin Heidelberg, 2007. LNCS 4727, Springer-Verlag.

[12] Reouven Elbaz, Lionel Torres, Gilles Sassatelli, Pierre Guillemin, Michel Bardouillet, and Albert Martinez. A parallelized way to provide data encryption and integrity checking on a processor-memory bus. In *DAC 2006*, pages 506–509, San Francisco, California, USA, July 24-28 2006. ACM.

[13] Blaise Gassend, G. Edward Suh, Dwaine Clarke, Marten Van Dijk, and Srinivas Devadas. Caches and hash trees for efficient memory integrity verification. In *In 9th Intl. Symp. on High Performance Computer Architecture*, 2003.

[14] A. Gonsalves. Hackers report breaking Phone 2.0. InformationWeek, March 12, 2008.

[15] W. Eric Hall and Charanjit S. Jutla. Parallelizable authentication trees. In *Selected Areas in Cryptography*, pages 95–109, 2005.

[16] Guerney D. H. Hunt. Secure processors for secure devices and secure end-to-end infrastructure. Available at `http://www.research.ibm.com/jam/secure-processors5-30-06.pdf`.

[17] IBM. IBM 128-bit processor local bus version 4.7, 2007.

[18] Goce Jakimoski and K. P. Subbalakshmi. On efficient message authentication via block cipher design techniques. In *Advances in Cryptology – ASIACRYPT 2007*, volume 4833, pages 232–248, 2007.

[19] L. Keliher and J. Sui. Exact maximum expected differential and linear cryptanalysis for two-round Advanced Encryption Standard. IET Information Security , Vol. 1, No. 2, pp. 53-57, June 2007.

[20] Hugo Krawczyk. LFSR-based hashing and authentication. In *Advances in Cryptology – CRYPTO 94*, pages 129–139, London, UK, 1994. Springer-Verlag.

[21] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. RFC2104 - HMAC: Keyed-hashing for message authentication. `http://www.faqs.org/rfcs/rfc2104.html`. Retrieved Sept. 17, 2008.

[22] David Lie, Chandramohan A. Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John C. Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. In *Architectural Support for Programming Languages and Operating Systems*, pages 168–177. ACM, 2000.

[23] Ralph Merkle. *Secrecy, authentication, and public key systems*. PhD thesis, Stanford Univeristy, 1979.

[24] Kazuhiko Minematsu and Yukiyasu Tsunoo. Provably secure MACs from differentially-uniform permutations and AES-based implementations. In *FSE*, pages 226–241, 2006.

[25] Kaisa Nyberg. Differentially uniform mappings for cryptography. In *Advances in Cryptology – EUROCRYPT 93*, volume 765, pages 55–64. Springer-Verlag New York, Inc., 1994.

[26] Douglas R. Stinson. Universal hashing and authentication codes. In *Advances in Cryptology – CRYPTO 91*, pages 74–85, London, UK, 1992. Springer-Verlag.

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

[28] Gookwon Edward Suh. *AEGIS: A Single-Chip Secure Processor*. PhD thesis, MIT, 2005.

[29] Trusted Computing Group. *TCG Specification Architecture Overview*, revision 1 edition, July 2007.

[30] Romain Vaslin, Guy Gogniat, Eduardo Wanderley Netto, Russell Tessier, and Wayne P. Burleson. Low latency solution for confidentiality and integrity checking in embedded systems with off-chip memory. In *ReCoSoC*, pages 146–153, 2007.

[31] M. N. Wegman and J. L. Carter. New hash functions and their use in authentication and set equality. In *J. Comput. System Sci. 22*, pages 265–279, 1981.

[32] Eric W. Weisstein. "Universal hash function." From MathWorld–a Wolfram web resource. `http://mathworld.wolfram.com/UniversalHashFunction.html`. Retrieved Sept. 17, 2008.