Inspection Resistant Memory: Architectural Support for Security from Physical Examination

Jonathan Valamehr[†], Melissa Chase[*], Seny Kamara[*], Andrew Putnam[*], Dan Shumow[*], Vinod Vaikuntanathan[‡] and Timothy Sherwood[†]

[†]UC Santa Barbara [*]Microsoft Research [‡]University of Toronto
valamehr@ece.ucsb.edu, {melissac, senyk, anputnam, danshu}@microsoft.com, vinodv@cs.toronto.edu, sherwood@cs.ucsb.edu

Abstract

The ability to safely keep a secret in memory is central to the vast majority of security schemes, but storing and erasing these secrets is a difficult problem in the face of an attacker who can obtain unrestricted physical access to the underlying hardware. Depending on the memory technology, the very act of storing a 1 instead of a 0 can have physical side effects measurable even after the power has been cut. These effects cannot be hidden easily, and if the secret stored on chip is of sufficient value, an attacker may go to extraordinary means to learn even a few bits of that information. Solving this problem requires a new class of architectures that measurably increase the difficulty of physical analysis. In this paper we take a first step towards this goal by focusing on one of the backbones of any hardware system: on-chip memory. We examine the relationship between security, area, and efficiency in these architectures, and quantitatively examine the resulting systems through cryptographic analysis and microarchitectural impact. In the end, we are able to find an efficient scheme in which, even if an adversary is able to inspect the value of a stored bit with a probabilistic error of only 5%, our system will be able to prevent that adversary from learning any information about the original un-coded bits with 99.9999999999% probability.

1. Introduction

Computer architects are asked to balance performance with all of the other aspects of the design including reliability, power consumption, cost, ease of use, and of course security. Just as we have developed architecture-level schemes to help manage circuit-level problems for the purposes of achieving reliability (dealing with soft-error susceptibility [51, 29, 5] and early wear-out [31, 41, 20] for example), new architecture-level schemes are needed to deal with the circuit-level security problem of information leakage through hardware examination.

Underlying many, if not most, modern security schemes is the idea of a key – a small set of bits whose secrecy ensures the effectiveness of the overarching policy. There are many architecture-level techniques by which these bits can be kept secret. Processors enforce the operating system's memory access policies, information flow aware hardware can prevent secret data from escaping at run-time [40, 45, 11, 46, 33], and even many side-channels can be mitigated through randomization or partitioning [47, 48, 42]. However, providing secrecy becomes even more challenging when the device in question is portable, such as a smart card or a cell phone, or when the keys are shared across many different devices, such as in many non-network based attestation schemes. In these cases, one needs to consider the fact that an adversary may be able to physically dismantle and inspect the system, bypassing all of our traditional security mechanisms.

If the secrets are of sufficient value, a determined attacker can use many different intensive methods to learn these bits from even minute physical differences imprinted on the chip by the storage of that bit. With tools available for rent at any major university or fabrication facility, a memory array can be carefully sliced apart with a focused-ion beam and inspected under an electron microscope. If, for example, the memory array was composed of anti-fuse memory cells [10], the attacker could slice the chip apart and observe which anti-fuses have been tripped. The nature of this class of attack is particularly insidious because it can be done when the hardware is not even powered on, leaving active defenses more or less useless. The problem is further exacerbated when the exact same secret is stored across an entire class of devices, allowing an attacker to piece together the full secret even if just a few bits are leaked from each device.

The goal of this paper is to examine ways in which we can architect memory structures that present the same interface as a simple memory, yet are hardened against these direct and destructive physical attacks in their most general form. If an attacker can perfectly deconstruct and read (with no measurement error) every bit in the system, there is little that can be done to protect the secret key – the entire state of the machine can be perfectly reconstructed by the attacker. However, the attacker's role is not so easy. Often there is error inherent to the process of measurement, for example, the attacker may fail to see the exact point at which the anti-fuse has triggered, leading them to believe that a zero was in fact a one, or vice versa. As these errors are made and some fraction of the bits are incorrectly identified, information is lost by the attacker.

Instead of storing the secret key itself on-chip, we propose that the keys (or other secret data) can be encoded using additional bits, increasing the number of bits that an attacker must identify in order to recover the secret key. The core of the idea is to force the attacker to successfully identify a large number of the encoded bits to be able to learn *even a single bit of information about the secret key*. The main idea of our approach is inspired by recent theoretical cryptography results [32, 7, 8, 12] that are themselves rooted in a subject familiar in computer architecture: error-correcting codes. In an error-correcting code, if we are able to recover at least k of the n bits measured, then the full message can be perfectly recovered. The idea we need here is exactly the converse, if less than k bits of the n bits measured can be recovered, then *no* information about the message is recovered (in an information-theoretic sense). This is embodied in the notion of secret-sharing first introduced by Shamir [35] in 1979. In fact, as we will discuss later in Section 3, there is an important relationship between error-correcting codes and the problem of secret hiding.

Resistance to physical attacks is a fundamentally new design constraint for computer architects, and there are some important new tradeoffs to consider. The biggest tradeoff we explore here is between area (the size blow-up due to both the encoding and the hardware implementation of the standard memory abstraction) and inspection resistance (the probability that the attacker will successfully recover the bits of the secret). We explore this design space, and make use of two related theoretical methods from cryptography originally developed for secret-sharing, a method for splitting data apart between multiple, potentially adversarial, parties. The memory architectures that result from these cryptographic primitives differ radically both in terms of failure mode and implementation costs. We further show how these two schemes can be combined together to create a balance between storage cost and computational complexity, and analyze how and when the resulting system can fail to provide security. In particular, this paper makes the following contributions:

- We propose an architecture-level model for estimating a memory system's vulnerability to physical inspection and memory remanence attacks, along with a discussion of the strengths and weaknesses of said model.
- We describe a new quantifiable architectural tradeoff between physical inspection resistance and hardware overhead, and we show that secret-sharing techniques can result in systems being built that are both formally sound yet implementable in practice under this set of tradeoffs.
- We evaluate several different memory structures both in terms of their ability to thwart these attacks and in their implementation cost. In the end we unify secret sharing and a coding based approach under a common framework, and describe a specific implementation that, even if an attacker is able to learn the value of physical bits with 95% accuracy, has less than a 1 in a billion chance of leaking even a single bit of the secret.

We begin by discussing our motivation for investigating how to store secrets on a device where the attacker has full access. We go on to discuss some background on memory remanence attacks and other physical attacks on cryptography as well as the scenarios in which they are problematic in real life, and a formalization of our threat model. With this background, we present the design choices for our architecture and analyze the tradeoffs between our different approaches.

2. Background and Motivation

To help ground these requirements in a real world context, let us consider the problem of network-less attestation for console gaming systems. To help prevent cheating as well as unauthorized and malicious knock-offs, a console system may wish to ensure that it talks only with certified devices. Before the console will talk with a new device, the device must first be able to "prove" in some way that it can be trusted, typically by demonstrating that it knows some secret.

These secrets may be symmetric and shared between all parties, but more often the secrets are private keys used in a public-key authentication method. Authentication can happen even without network access as long as both parties involved can convince each other that they know what that shared secret is. There are well known cryptographic methods, such as public key authentication, that allow this process to occur without actually requiring the *transmission* of any secrets. However, each system must, in some form or another, physically possess and make use of that secret. The problem is that the cost of leaking that key is now *much* larger. Once the secret is leaked, any number of new systems can be created containing that same key and the scheme is completely broken. Such "break-once run-anywhere" attacks provide tremendously attractive and potentially lucrative targets for attackers. Often, such attackers are often willing to bring highly specialized equipment and training to bear [18]. While the applicability of our approach is not limited to such extremely high value data, it is certainly one motivating instance.

2.1 Physical Inspection Attacks

The problem of storing a secret in a device when the attacker has full physical access to that device is extremely difficult. At a fundamental level, attacks which attempt to infer a set of bits from a device can be divided into two classes: *passive attacks*, in which the interface of the system is probed for either timing or electrical differences, and *intrusive attacks*, those that actually breach the boundary of the package, allowing the attacker to scan, probe, or alter the physical hardware itself. While still a topic of further research, passive attacks and their countermeasures are discussed extensively in prior work. In this paper we concern ourselves primarily with the latter, more intrusive style of attacks.

Intrusive attacks can then be further subdivided into either: powered attacks, meaning that the device is monitored in an active (powered-on) state as it attempts to perform it's intended functionality; or un-powered attacks, meaning that the bits are extracted from the device while the hardware is not even powered on. The classic approach to dealing with powered attacks is to insert some form of tamper-detection into the device, but preventing attacks when the system is powered off is even harder because the system (in its powered off state) has no computational ability to react to changes in the environment. One way around this problem is to include a battery or other source of power within the boundary of these sensors [37]. In this way the system is never truly powered off, and it is always ready to react when the surface of the platform is breached. Unfortunately, this means that a power source must be included in the tamper-detecting package, which greatly increases the total size of the package, the amount of surface that needs to be protected, and hence the total cost of the design. While this provides a platform secure against almost all intrusive physical attacks, there are two main problems here: 1) finding a reaction to intrusion that *completely* destroys that data, and 2) the cost of the system (often well outside the set of costs acceptable in the embedded device space where even a few cents matters). After the countermeasures are deployed, the device can still be depackaged, at which point it would still be subject to further unpowered attacks.

Independent of whether active counter measures have been applied or not, we specifically consider unpowered attacks where the attacker is free to slice, cut, and examine the silicon in any way that they desire. Specifically the threat model that we consider is one in which an attacker is given physical access to a dead device on which a secret key is stored or has been stored (Figure 1).

Information about that secret could exist in many different forms. The typical way that these attacks are carried out is that the silicon is cut into slices with a Focused Ion Beam (FIB) and examined under an electron microscope. Most architects have probably seen such silicon cross sections in other contexts. The devices are examined one by one and tell-tale signs of current fatigue, such as whisker tails due to electron migration, are recognized by an expert. While the equipment involved is quite expensive to purchase outright, most companies and campuses rent out their machine by the hour. Indeed, there are teams of people that specialize in carrying out this sort of analysis for post-mortems (e.g., when digital systems are involved in violent accidents) and reverse engineering. Consider the remanence issues with three common memory technologies:

Anti-Fuse: Anti-fuses are a non-volatile, write-once memory that are formed by creating an electrical connection through the application of high current across a thin channel. The actual connections are electrically very stable, yet are very difficult to examine even under a scanning electron microscope. Their security properties have been primarily analyzed within the context of FPGA reverse engineering [10, 49].

EEPROM and Flash: Flash memory and EEPROM are similar in structure, with both storing charge on a floating gate. When EEPROM and flash cells are overwritten, some residue of the previous bits written are still contained within the substrate as bias, and differences in the threshold voltage or gate voltage can be measured to detect the state of a cell. This effect is particularly noticeable in infrequently-programmed cells [24], which is likely the case for cells holding a secret key. Prior work has shown that this bias can survive even tens of redundant "clean-up" writes, making complete erasure of the information difficult within the time dictated by a countermeasure[17].

SRAM: Even volatile memories are subject to analysis in many cases. SRAM, while storing a bit for even as little as a half a second, can develop tell-tale signs due to differences in the electromigration at the bit-cell level [36]. Because of this, even volatile memories that hold secret keys need to be protected.

While physical analysis gives an attacker incredible access to the internals of the system, we believe these analyses cannot be 100% accurate. If the bits are stored unprotected, even unreliable approaches that can determine as few as 25% of the bits stored in memory are sufficient to inform further analysis to the point that the cryptographic schemes (e.g., network authentication protocols) using the key can be broken. However, as we will show later, even if these techniques were to be 95% accurate, it is possible to

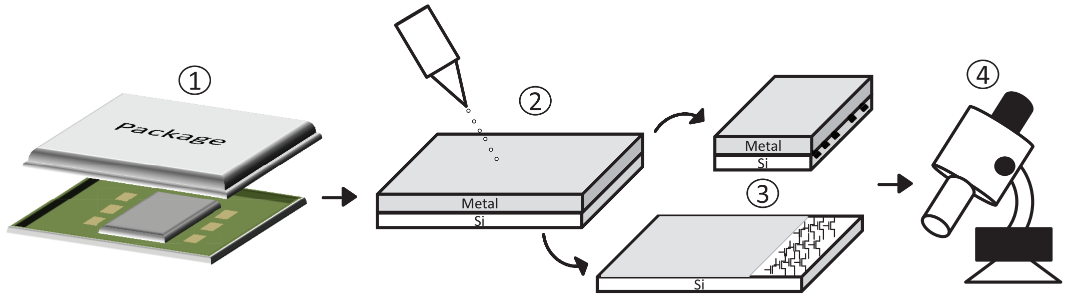

Figure 1: An overview of the general steps taken to conduct an unpowered, physically-intrusive attack. In step 1, any packaging is disassembled to expose its contents, and in step 2, the silicon chip is obtained. Step 3 uses a Focused Ion Beam to either cut a cross section of the chip, or to slice off layer after layer until individual transistors are exposed, Finally in step 4, an electron microscope is used to examine the chips and retrieve the desired data.

create reasonable architectures that prevent even a single bit of the key from leaking.

2.2 Other Physical attacks and Related Work

It is worth pointing out the other attack models relevant in this space and the best known methods for addressing them. There are many channels by which a secret may leak to an adversary beyond direct inspection. Many of the powerful attacks rely on observations of the dynamic behavior of the system to infer information about the keys. For example, information about a key can be observed through variations in timing due to cache misses [43, 30, 4], variations in power utilization over time [23], or through RF emanation [14]. Instruction caches, data caches, and branch predictors are some of the shared resources that can be exploited in these attacks [22, 2, 1]. All of these channels are powerful in the hands of a motivated attacker, but there is a growing body of work attempting to develop strong countermeasures for general models of these attacks. We view these attacks as orthogonal to the problem of direct measurement addressed in this paper.

In response to the success of side-channel attacks, the theoretical cryptography community has recently developed rigorous models of information leakage and cryptographic schemes that can be proven to retain security in these models. Although roots of this work can be traced back to early work on privacy amplification, the problem of information leakage was explicitly considered starting from the works of Rivest [32, 7, 8, 12], and developed in its full generality by Micali and Reyzin [28] as well as Ishai, Sahai and Wagner [19]. This sub-field of theoretical cryptography has come to be known as “leakage resilient cryptography”, and the amount of interest and the number of results in this area has exploded in recent years [13, 3]. Although there have been a host of interesting results in this area recently, most of the schemes are rather inefficient, and restricted to particular cryptographic mechanisms such as encryption or authentication. We will, however, use the intuition and techniques from these works, and in particular the work on exposure-resilient cryptography by Dodis et al. [8], to construct an inspection resistant memory architecture that is efficient enough that the end system can be used in a real design. Finally, we remark that the focus of many constructions in this area is to protect computation from leakage attacks, whereas in this work, we focus exclusively on protecting memory components from a very powerful form of inspection.

Another area that is related to the work presented is tamper resistant computing. Architectures in this space generally attempt to make any tampering with memory or the bus evident through the use of sophisticated hashing and encryption schemes [39, 25, 16, 26]. Yang and Peng extend this idea to consider the protection of keys by secret sharing them across multiple cores in a CMP [50]. We believe that our approach is complementary to these architectures. The threat model these systems typically address is one in which the entire memory subsystem is adversarial, but they often assume that the processor itself is a valid root of trust, an assumption that would be assisted by the technology proposed here.

A final area of related work is Physical Unclonable Functions (or PUFs). The goal of a PUF is to provide a function that is intimately tied to the underlying physical hardware and yet is intractable to duplicate. This, in turn, allows hardware to be uniquely identified through a series of challenges and responses [38, 6, 27, 15]. A PUF does not provide secrecy in itself, rather it provides a type of one-way function with exponential complexity in linear space. There has been a flurry of work over the last several years exploring ways in which process variability can be exploited to provide the randomness required to build strong PUFs, and a good overview of that work along with other applications can be found in “Towards Hardware-Intrinsic Security” [34]. Many proposed implementations use PUFs to generate a unique key which is then stored in memory and, as such, may be subject to the types of attacks discussed in this paper. The work presented here is complementary to

PUFs, both in that it does not require any particular process variability assumptions or special fabrication techniques, and because the techniques may be helpful when used in tandem (for example after key derivation or in the assistance of SRAM-based PUFs).

2.3 Architectural Goal and Attack Model

We wish to create a hardware component that will: a) Act like a standard memory, allowing random access to the stored keys with no special restrictions; b) Allow some distribution of bits to be learned by an adversary without giving away any information about the secret keys; and c) Be efficient enough that the end system can be included in a real design.

Our secret-store memory keeps some number of keys, each of length k. Each key is stored in an encoded form that takes some larger number of bits c; we will call this the "encoded key". An attacker examines the c bits of the encoded key through whatever methods are at their disposal, and they make a best guess as to the state of each of the bits of the encoded key. The attacker, when dissecting and analyzing each bit has a probability p of learning the correct value of the bit, and a probability $1-p$ of learning nothing about the bit. In particular, as a running example, we will consider the cases where $p = 90\%$ which, by this reasoning, is at least as strong as saying that the adversary guesses correctly with probability 95% and incorrectly with probability 5%. In this paper, we do not make any estimation as to what a reasonable value of p may be, but rather present results and the architectural impact across the spectrum of possibilities.

We will be very conservative, and consider an attack successful if the attacker can infer *anything* about the secret, even a single bit. Specifically we define the probability that an attack is successful P_{succ} as the probability that given an architecture and a probability p of successfully attacking an individual stored bit, any information leakage has occurred. While a single bit leakage is not enough to break any reasonable cryptographic scheme (because an attacker could have just tried both possible options of 0 and 1), if other copies of the key are known to exist an attacker might combine information from many such attacks to reduce the keyspace far enough that it can be searched exhaustively.

There are two potential issues with this model of an attack that are worth mentioning. The first is that, for the purpose of analysis, we assume that the probability an attacker learns anything from one bit is independent of the probability of the other bits. While we have no evidence to suggest that this assumption is invalid, it would be easy to modify the analysis to consider any correlation effects that did appear or, if necessary, to conservatively use the worst-case probabilities. The second issue with this model is that it ignores the ability of an adversary to learn anything from worn *logic*. While this is often less of a concern in practice because logic outputs tend to change more frequently, this is far from a given. However, such analysis is beyond the scope of this work.

3. Architectures

There are several different architectural options here, and we begin with the simplest, and work our way quantitatively towards the best possible options. While we build on some different cryptographic primitives, in each case we believe that we are the first to propose any of these schemes in the context of physical inspection based attacks. Furthermore, we are the first to quantify the costs of these schemes in a practical hardware sense (i.e. with more than simple asymptotic bounds). For the purposes of analysis, we will assume (unless otherwise stated) that the key-length required is $k = 1024$, the probability of successfully attacking a single stored bit $p = 90\%$, that the store has a total capacity of 128 keys, and that an acceptable value for P_{succ} is 1 in a billion (although we will show later that the analysis is *surprisingly* unaffected by the exact value of P_{succ} that we choose as "acceptable").

k	Key Length
p	Probability of learning 1 bit
P_{succ}	Probability of successful attack
x	1 secret bit
s	Number of random bits
c	Total number of bits to be stored per key
r	1 random bit
T	Random bit s-by-k matrix

Table 1: A list of terms and variables used in the upcoming section

3.1 Option 0: Do nothing

To provide a base line for our discussion, the first option we have for hiding the secret is to store it *as-is*. As outlined in Figure 2, the chance that an adversary learns something about the key is obviously very high. With k bits to attack, and a probability p of successfully attacking an individual bit, the chance that the adversary *fails to uncover even a single bit* of the key is exceedingly small, namely $(1-p)^k$. The number $(1-p)$ – namely, the probability that the adversary does not learn an individual bit – must be extremely close to 1 for the exponent to not drive the entire quantity to 0. In fact, in expectation, the adversary obtains not just a single bit, but $p \cdot k$ bits of the key (out of a total of k bits) in this scenario.

3.2 Option 1: Apply the Idea of Secret Sharing

The first, and easiest to understand, option for hiding a bit of information is to use secret sharing scheme. A simple and efficient secret sharing scheme consists of XORing the

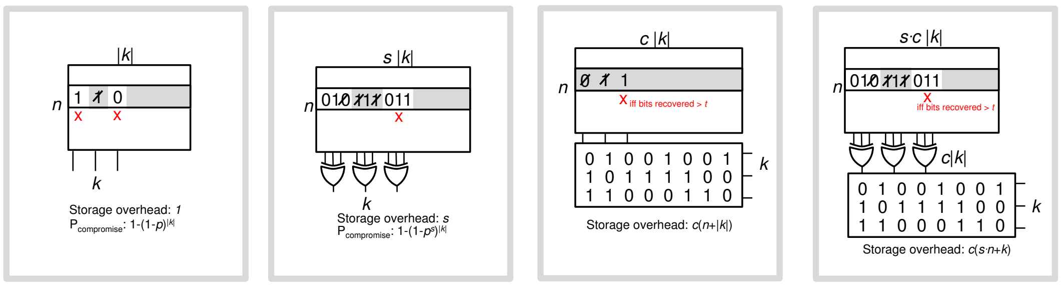

Figure 2: An overview of options 0 through 3 discussed below. The first (left most) option is to do nothing, which an attacker can break very easily. Secret sharing divides the bits up in a way that XOR is used to put them back together, but requires a significant amount of space. The next option is to use a code (based on multiplication by a binary matrix) to protect the key, which is better but requires the storage of a large matrix. The final (rightmost) option presented here is a combination of the two that seeks to strike a balance between matrix size and code density.

secret x with s random bits $(x_1, \ldots, x_s)$, and setting the shares to be $\left[x_1, x_2, \ldots, x_s, x_{s+1} = x \oplus (x_1 \oplus x_2 \oplus \ldots \oplus x_s)\right]$. Of course, given all the $s+1$ shares, one can recover the secret x by computing $x_{s+1} \oplus (x_1 \oplus x_2 \oplus \ldots x_s)$. On the other hand, if the adversary obtains at most s of these $s+1$ shares, he has *no information* about the secret x. Namely, no matter how powerful the adversary is, they do not learn *any information* about the secret x as long as they do not get all the shares $x_1, x_2, \ldots, x_{s+1}$.

Intuitively, one can think about secret sharing as described in Figure 2. To share a 0 bit three ways, we first take two random bits. We then set the third bit such that the parity of the three bits is 0. A change (or, measurement error) in *any* of the three bits will result in a change in (or, measurement error) in value stored. An astute reader will see the relation with error codes start to take shape here. In essence, a secret sharing scheme uses the *parity* bit as the data, and a set of randomly selected "data" bits as the "code". The fragility of the parity bit is then used to catch errors in one scheme, but to hide data in the other.

How do we store a k-bit key using this method? The simple option is for each individual bit of the key to be shared across $s+1$ stored bits, increasing the total number of bits to be stored per key to $c = (s+1) \cdot k$. To be successful, an attacker must compromise *all* of the bits of *at least one* $(s+1)$*-bit share block*. The probability of compromising a share block is now p^{s+1}, and our attacker has k independent trials to succeed, yielding a total probability of success $P_{\mathsf{succ}} = 1 - (1 - p^{s+1})^k$. Figure 3 explores the relationship between the number of shares s needed, and P_{succ}, the probability that the attacker learns at least one bit of the actual key for the parameters discussed at the beginning of this section. Interestingly, there is a steep drop in the probability of a successful attack as we grow from 40 to 80 shares per bit. Clearly, replicating every bit in the key by a factor of 80 is a steep overhead to endure, but as we increase s we can drive P_{succ} arbitrarily close to zero.

3.3 Option 2: Encode with a Random Matrix

Intuitively, the reason that the secret sharing based scheme has such a large overhead in storage is that we chose to encode *each bit separately*. This raises the possibility of *encoding all the bits together* and achieving much better parameters. A coding based scheme is a natural extension of the secret sharing scheme yet shares many of its theoretical properties [12]. Recall that in the secret sharing scheme, we encoded a bit x by picking s random bits $x_1, \ldots, x_s$ and computing $x_{s+1} = x \oplus (x_1 \oplus \ldots \oplus x_s)$. To encode a second bit, we picked a new, fresh sequence of s random bits, and so on. A natural question that comes up is: *can we reuse the same sequence of random bits $x_1, \ldots, x_s$ to encode multiple bits?* If yes, how many bits can we possibly encode using a single sequence of random bits? Answering these questions leads to a coding based scheme which provides (close to) the best trade-off between the amount of storage needed to encode k bits and the probability of success P_{succ}, for a given leakage rate p.

Before we proceed to discuss this solution in detail, let us reflect on the intuition for what are the best parameters we can possibly hope to achieve using such a scheme. Note that in expectation, the adversary obtains a p fraction of the c stored bits, or equivalently, an "information content" of $p \cdot c$ bits. Thus, the "residual entropy" left on those c bits is $(1-p) \cdot c$ bits. To encode k bits in such a way that the adversary obtains absolutely no information about any of them despite the leakage, we need *the residual entropy to be larger than the length of the key*. That is, we need $(1-p) \cdot c \geq k$. In other words, the total number of bits of storage required to *perfectly protect* k bits of the key against a rate of leakage of p has to be at least $c \geq \frac{k}{1-p}$ bits - this is a strict lower bound, and it is impossible to do any better and still guarantee security in our model. For our parameters of $k = 1024$ and $p = 90\%$, this translates to a storage of $10 \cdot 1024 = 10240$ bits. In fact, we will now show a scheme that achieves storage very close to this number, while keeping P_{succ} very small.

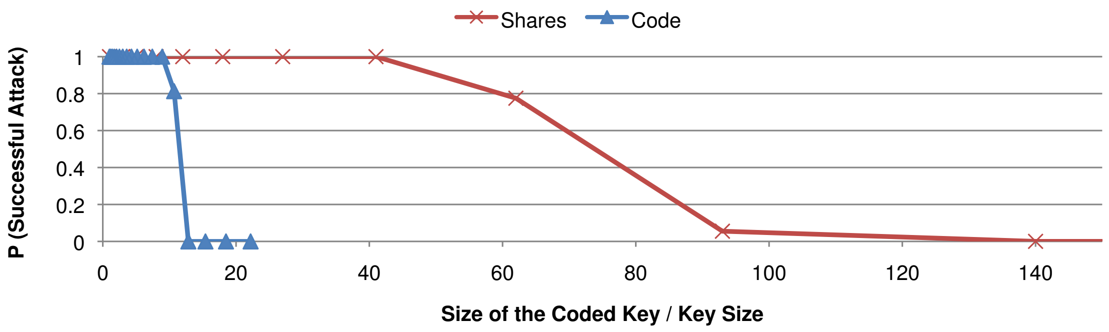

Figure 3: A graph of the tradeoff between code size and attacker success probability. On the x axis we track the size of the coded key, where either secret sharing ("Shares"), or code theory ("Code") is used. On the y axis is the probability that an attacker will be learn at least one bit of the secret (lower is better). Because secret sharing can be broken by an unlucky distribution of errors concentrated on a single coded bit, far more bits are need to encode the key to reach the same attacker success probability.

At a high level, the coding based scheme works as follows. We will encode the key as s random bits plus k "data bits". In order to do this, we fix random subsets $T_1, T_2, \ldots, T_k \subseteq \{1, 2, \ldots, s\}$ in advance. The choice of these subsets do not depend on the data to be encoded, and in fact, in the implementation, they will be chosen at random once and for all, and stored on-chip. To encode a sequence of k bits $x_1, \ldots, x_k$, we choose s random bits $r_1, \ldots, r_s$ (as before) and compute the k "data bits" r_{s+j} for $j = 1, \ldots, k$ as $r_{s+j} = x_j \oplus \left(\bigoplus_{i \in T_j} r_i\right)$. In other words, we compute the XOR of a subset of the s random bits and XOR the result to the bit that we wish to encode. The final encoding is computed by the formula:

$$\Big[r_1, \ldots, r_s, r_{s+1} = x_1 \oplus \bigoplus_{i \in T_1} r_i, r_{s+2} =$$

$$x_2 \oplus \bigoplus_{i \in T_2} r_i, \ldots, r_{s+k} = x_k \oplus \bigoplus_{i \in T_k} r_i\Big]$$

It is worth noting that the savings in space comes from the fact that unlike the secret sharing scheme, we are re-using the random bits $r_1, \ldots, r_s$ to encode all k bits of the key.

Instead of carrying around this big formula, let us write it out as a simple matrix-vector multiplication. Write each subset T_j as an s-bit column vector whose i^{th} bit is 1 if and only if $i \in T_j$. Thus, a subset $\{1, 2, 5\}$ is represented by the (column) vector $(1, 1, 0, 0, 1, 0, \ldots, 0)$. Let T be the random s-by-k matrix whose *columns* are all these s-bit vectors, one corresponding to each subset $T_1, \ldots, T_k$. Also, let the vector $\vec{r} = (r_1, \ldots, r_s)$, and let the vector $\vec{x}$ be $(x_1, \ldots, x_k)$. Now, we can concisely write the encoding computed above as $\left[\vec{r}, \vec{r} \cdot T \oplus \vec{x}\right]$, with storage equaling $c = s + k$ bits.

Thus, encoding a string of k bits consists of (a) choosing s random bits, (b) computing a matrix vector product of these s bits (written out as a vector) with a *fixed* s-by-k matrix, and (c) XORing the result with the k-bit key to be protected. Decoding proceeds by a symmetric sequence of operations – first, compute the matrix-vector product $\vec{r} \cdot T$, and then "XOR this out" from $\vec{r} \cdot T \oplus \vec{x}$ to get the key $\vec{x}$. A downside of this scheme is that the matrix T needs to be chosen at random and it needs to be stored on chip. However, the storage required for the matrix can be amortized if we store a large number of keys, since we can re-use the same matrix for all the keys. This effect can be seen in Figure 6. We will present another option to significantly reduce the overhead of storing the matrix in Section 3.5.

We now derive an expression for the success probability P_{succ}, for a given storage overhead s and a per-bit leakage probability p. The calculation of this probability proceeds in two steps. First, we show that with a very high probability, the number of bits that the adversary learns is in fact not much larger than $p \cdot s$. (Recall that the expected number of bits that he learns is $p \cdot s$.) Thus, whenever the adversary learns significantly more than $p \cdot s$, we say that he automatically succeeds, and the probability that he learns significantly more than $p \cdot s$ bits contributes to P_{succ}. Notice that this calculation overestimates the adversary's probability of success (although not by a significant margin). Analytically, the probability that the adversary learns more than, say, $(p + 0.01) \cdot s$ bits turns out to be (at most) $2^{-2 \cdot (0.01)^2 ps}$, using a Chernoff bound [9]. Secondly, given that at most $(p + 0.01) \cdot s$ bits leak, the probability that the adversary obtains information about *any* of the key bits can be shown to be at most $\frac{2^{k+1}}{2^{(0.99-p)s}}$, using an argument based on matrix rank. Thus, in total, an upper bound on the adversary's probability of success is

$$P_{\mathsf{succ}} \leq 2^{-2 \cdot (0.01)^2 ps} + \frac{2^{k+1}}{2^{(0.99-p) \cdot s}} \qquad (1)$$

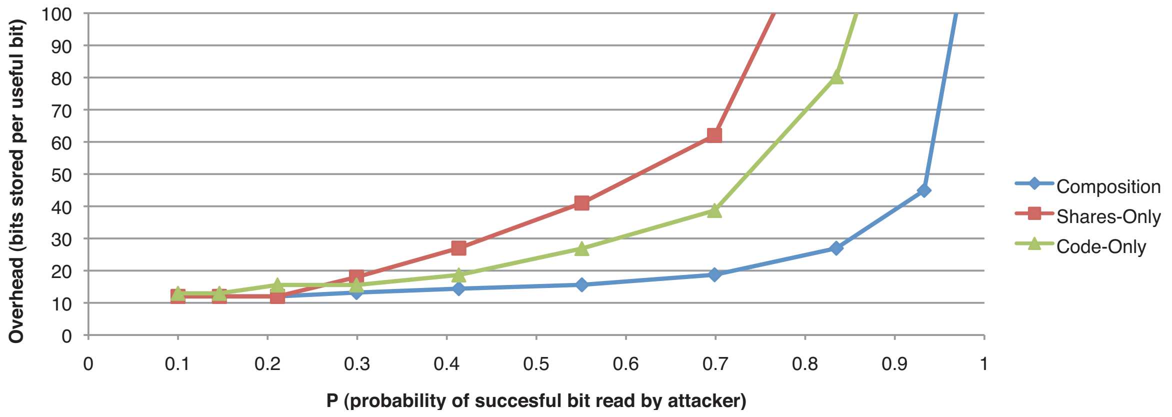

Figure 4: A graph showing the efficiency of the different schemes as a function of the power of the attacker to read individual bits. The x axis is the probability p that an attacker is able to correctly determine the value of each stored bit in the scheme. The y axis is the size of the code (as a ratio) needed to ensure that the probability that an attacker can learn at least one bit is below one in a billion. While the code scheme scales better as the attacker becomes more powerful, the combination of the two methods outperforms either individual approach when the matrix needs to be stored on chip.

For our purposes, the dominant term in this sum is the second one. We can see that it drops off sharply as soon as the exponent of 2 in the denominator starts exceeding the exponent of 2 in the numerator. That is, as soon as $s \geq \frac{k+1}{0.99-p} \approx \frac{k}{1-p}$, as predicted by the informal analysis above. For our scheme, the storage required for the encoded key is $s+k$ bits (s bits of randomness plus k bits of the XORed key), which is quite close to the optimum of $\frac{k}{1-p}$. From Figure 3, we see that for our choice of parameters $p = 90\%$ and $k = 1024$, the point where P_{succ} starts to fall off is when the ratio of the coded key versus the actual key length, i.e., $(s+k)/k$ is approximately 11 or 12, which coincides with this analysis.

Another interesting observation from Figure 3 is that the drop-off in P_{succ} is *much steeper* for the coding based scheme than the secret sharing scheme. Informally, the reason is that the secret sharing based scheme encodes each bit separately. Thus, the probability of learning i bits out of k drops off roughly *linearly* with i from its largest value to the smallest. This explains the gradual incline for the secret sharing based scheme in Figure 3. For the coding-based scheme, as long as the adversary does not learn too many bits of the encoded key, they obtain *no information* at all about the actual key. As soon as they learn more than a threshold number of bits of the encoded key, they start learning a lot of information about the actual key all at once. In other words, as we make the encoded key larger (for a fixed value of p), there comes a point when the attacker learns so few bits (compared to the length of the encoded key) that he has no information at all about the actual key. This explains the sharp drop-off in P_{succ} in Figure 3 for the coding scheme.

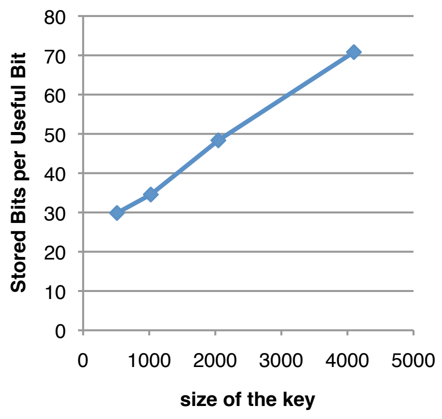

Figure 5: A graph showing the efficiency of the combined scheme as a function of the size of the key. As the key grows beyond 1024 bits, we can see that the efficiency of the scheme decreases (as the number of stored bits required per key bit increases linearly)

3.4 Option 3: A New Combination

Both the options presented above – the secret sharing based solution and the coding based solution – have pros and cons. On the one hand, the secret sharing based solution has a *larger storage overhead for the key* due to the fact that the scheme squanders away random bits. But, the scheme does not need to store anything beyond the encoded key. On the other hand, in the coding based scheme, the overhead to store the key is relatively small, but we also need to store a fairly large random matrix T. While both those schemes have been previously proposed for other applications, we believe we are the first to quantify their actual implementation overhead, and in doing so realized a natural option is to design a novel hybrid scheme that achieves the best of both worlds.

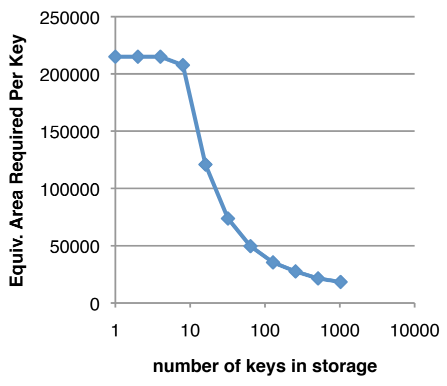

Figure 6: A graph showing the efficiency of the optimally combined scheme as a function of the number of keys stored in the architecture. With very few keys stored, the cost of the matrix cannot be amortized, and the best scheme uses secret sharing only. However, once we get beyond on the order of 10 keys stored, the efficiency of the scheme improves drastically as the cost of larger matrices can be amortized.

To do this, we first encode the individual bits of the key using a code-based secret sharing scheme with a smaller matrix T. Roughly speaking, the intent is to reduce the severity of attack from a per-bit leakage probability of p to a smaller number p'. We then further encode the resulting string using the secret sharing based scheme. The upshot is that the matrix needed for the coding-based scheme is smaller because it only needs to be strong enough to reduce the severity of the attack from p to p'. As shown in Figure 4, the combined scheme outperforms both the coding based and the secret sharing based schemes when the matrix T is stored on chip. Figure 5 shows how the storage overhead varies as a function of the length of the key. We remark that although the increase in the storage overhead with the length of the key is disturbing, it is predominantly due to the need to store the large matrix T, and will be removed by our improved solution in Section 3.5.

Figure 6 shows that the overhead caused by storing the matrix can be amortized by the number of keys we store in the system. This is simply because we can use the same matrix T to encode many different keys. Figure 7 shows a plot of the area required to store the encoded key (together with the auxiliary information such as the matrix T) as a function of the success probability of the adversary. Obviously, if we let the adversary succeed with probability 1, very little storage is required. The storage jumps to a certain number as soon as we demand the adversary's success probability to be less than, say, 0.1, and stays more or less constant from then on. This phenomenon can be explained by the (roughly) logarithmic dependency of the storage on the success probability. In other words, reading Figure 7 from right to left roughly gives a logarithmic curve.

3.5 Option 4: Dynamic Matrix Creation

In the last section, we showed a way to ameliorate the effect of storing the huge matrix T in the coding based scheme. We now show a way to eliminate this overhead almost entirely, using a cryptographic hash function such as SHA-2. The SHA-2 family of hash functions consists mainly of two functions SHA-256 and SHA-512 (where the numbers indicate the output length of the hash function). The advantage to this approach is that SHA-2 is widely used, and has optimized hardware implementations already available. In particular, most of the next generation SHA implementations require on the order of $20,000$ gates to implement, which is quickly amortized over the entirety of the key storage architecture.

Informally, the idea is to instead store some compressed version of the matrix T, and then decompress portions as necessary while we are regenerating the key. However, our security analysis above requires that the matrix be chosen completely at random, which inherently means that it cannot be compressed much. Alternately, we could try to modify the way we choose the matrix so that the result is easier to compress. This must be done carefully, because the wrong approach may result in a completely insecure scheme.

Our approach is to use cryptographic tools which are specifically designed to produce output that looks random in a very strong sense but that can be generated deterministically from a very small input. In particular, we note that modern cryptographic hash functions are designed to behave like truly random functions (in which each output is chosen independently and uniformly at random) as much as possible. (See e.g. the security requirements given by NIST for the SHA-3 competition.) Evidence of any non-trivial application in which a hash function behaved significantly differently from a random function would be considered a major weakness, and finding such a weakness in SHA-2 would be a major result.

So, suppose that we use SHA-256 as follows to generate our matrix: Recall that T is an $s \times k$ matrix of bits. Then we will generate each column using $s/256$ calls to SHA-256. Specifically, we will choose a short random string seed, and compute the ith column as: $SHA2(\text{seed} \circ i \circ 1) \circ SHA2(\text{seed} \circ i \circ 2) \circ \ldots \circ SHA2(\text{seed} \circ i \circ \ell)$ where $\ell = s/256$, seed is a 256-bit string, $\text{seed} \circ i \circ j$ is encoded as a 448-bit string to be input to the SHA-256 hash function, and $\circ$ denotes concatenation of strings. Note that if we had instead used a random function, then the resulting matrix would be random, and would guarantee that an adversary would have only a very small probability of extracting any information about the key (as described).

But what if an adversary could somehow use the fact that the matrix was instead generated using SHA-2 to design a

Single seed (anti-fuse memory)	0.044 mm^2
Single key (SRAM)	0.084 mm^2
SHA generator	0.46 mm^2
Computational logic	0.08 mm^2
Total (for 1 key)	0.589 mm^2
Total (for 128 keys)	6.18 mm^2

Table 2: The area overhead for our proposed method, in 65 nm process technology. Note that the contents of anti-fuse memory need not be secret as they are used to determine the public matrix used for coding

better attack on our scheme? The key idea here is that that attack would then be a simple example of an application where SHA-2 behaves very differently from a random function. While we cannot say for certain that this is impossible, it would certainly be a very surprising result, and a major breakthrough in the cryptanalysis community.

To ground the microarchitectural impact of this fourth option, consider the key length $k = 1024$ and the probability of successfully attacking a single stored bit $p = 90\%$. This fourth option requires 35 bits of additional storage per bit of the key, resulting in 35,840 bits per key. Using Cadence InCyte ChipEstimate 4.0 at a generic 65 nm process node, the total area required for storing the seed using anti-fuse memory is 0.044 mm^2. Note that we are not relying on the secrecy of these bits. Storing a key using the most area-efficient single-port SRAM configuration requires 0.084 mm^2 per key. The SHA generator requires 0.46 mm^2, and computation logic requires 0.08mm^2. In total, the cost of securing a single key using this method is 0.589 mm^2. Storing additional keys helps to amortize the cost of the SHA generator and the computation logic. Storing the 128 keys requires 6.18 mm^2, still well within the area bounds for embedded and consumer devices. This is summarized in Table 2. We evaluated the impact of this scheme in terms of area because we want to show the optimal amortization of code and hardware under the given tradeoffs. Simply expressing the overhead in terms of extra bits hides the cost of the hardware.

Thus, our cryptographic hash based scheme works as follows: We choose the 256-bit string seed – and use it to generate the matrix T on the fly, during the encoding and decoding procedures. The seed is then stored along with the encoded key. Note that instead of storing the huge matrix T, we now store the seed which is a 256-bit string. This architecture is sketched out in Figure 8. With this modification, the coding-based scheme is strictly better than the secret sharing based scheme in terms of total storage.

It is important to keep in mind the two caveats of this cryptographic hashing-based extension. First, the secrecy of the bits is based on the assumption that SHA-2 does in fact behave like a random function in the application we consider (this follows as a special case of the widely used

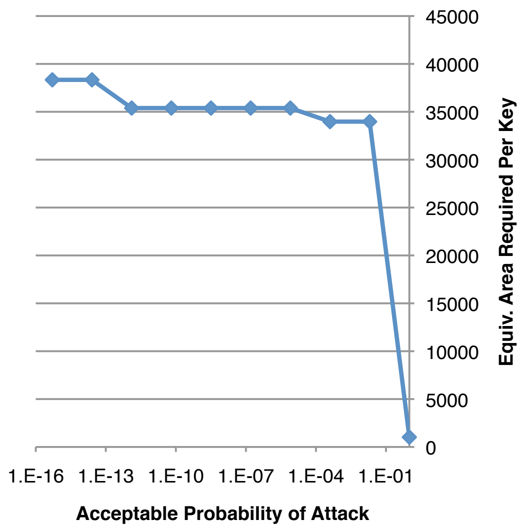

Figure 7: A graph showing the core size requirement as a function of acceptable attack probability. On the x axis we plot the risk probability that one might consider "acceptable". On the y axis is the size of the hardware required (in bits) for the best scheme. When we are willing to accept 100% probability of breaking then clearly very little space is needed. While there is a large jump as soon as we demand even a 1 in 10 probability of breaking, we can then demand up to 1 in 10^{16} (reading the graph from right to left) with only incremental increases in hardware overhead.

"random oracle heuristic"). Secondly, during the encoding and decoding process, we need to generate the matrix T on the fly which could potentially slow down the encoding and decoding algorithms. However, the computation of the matrix can be done in parallel with the encoding process and furthermore, these two processes can be perfectly pipelined to the extent that this computational overhead is barely noticeable.

4. Conclusion

Computer architecture has a long history of being at the forefront of technology, helping to bridge the hardware/software divide through more efficient implementations, cross-layer optimizations, and novel abstractions. One might ask if the contributions of this paper really are "computer architecture" as it certainly does not look like a paper from 20 years ago. There is no question that this paper relies on cryptographic techniques that are outside the background of many computer architects. However, while the tools are new, the goal is old – to create a new hardware abstraction, to encapsulate complexity, and to provide a building block for new software and systems to grow around.

Prior architecture contributions have considered the ability of designs to cope with hardware failures, errors in the memory hierarchy, and even early wear-out. They present

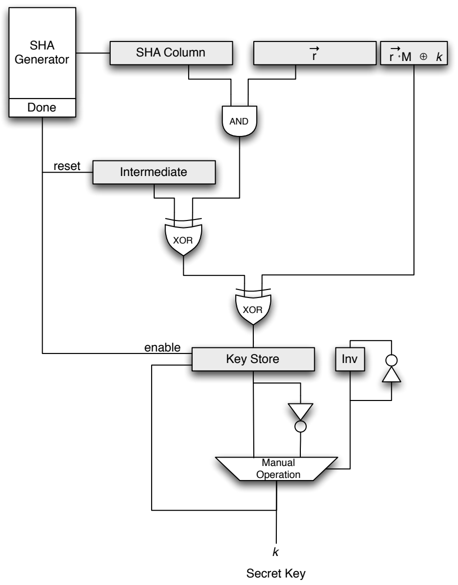

Figure 8: The best performing scheme combines the matrix multiplication operation of the code scheme, but does not store the actual bits of the matrix on chip directly, rather it generates them dynamically through the use of a cryptographic hash function. Most of the next generation SHA implementations require on the order of 20,000 logic gates to implement, which will quickly be amortized over the entirety of the key storage architecture. The key store SRAM is inverted on every cycle to prevent any signs of electromigration that the attacker could use to directly identify bits of the secret key.

new abstractions that attempt to hide these problems from users through careful design and novel analysis methods. We follow in this tradition by evaluating a novel set of hardware methods capable of abstracting away a *new* class of problems: physical memory inspection. When an attacker has unfettered physical access to a device and complex tools at their disposal it opens up significant new avenues for attack, and the hardware architecture needs to be involved in any attempt to make attack more difficult for the adversary. If a single secret is to be shared across many such devices, the amount of information that one can afford to leak from any individual chip will be exceeding small (not even a single bit). The minute physical differences in the memory circuitry caused by wear, improper or insufficient clearing, and/or the minute variations used to the store the bits themselves, are a prime example of such a leakage mode, and one that is not addressed easily with prior approaches. Of course this is not the only way in which the bits might leak; they may be exposed through timing, power, or EM emanation variations in addition to these more intrusive attacks. The methods presented here need to be used as a piece of a comprehensive strategy to manage these different channels.

Rather than tie our scheme to one particular type of examination error, we have generalized our analysis to consider any general uniform error p. If more is known about the distribution of errors for the particular memory technology and/or use scenario in question, the general techniques presented should still be applicable, albeit with slightly different equations governing the failure probabilities. Regardless of the distribution, no code of length c will be able to hide all the information about a key of length k if more than $c - k$ bits of the coded key are learned by an attacker. While this paper concentrates specifically on special inspection resistant memories, we see both the analysis methods and the memory block we provide as being a significant step towards a more general purpose inspection-resistant architecture. However, even if such a general purpose result is not possible, the memory abstraction we provide should still prove useful – we have shown that even if an attacker is correct in their analysis 95% of the time, our scheme can prevent the attacker from having any practical chance of uncovering information from the device with overheads that are diminishingly small with respect to an optimal solution.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments. This research was funded in part by National Science Foundation Grant CNS-0910734.

References

[1] O. Acıiçmez. Yet another microarchitectural attack: Exploiting I-cache. In *Proceedings of the First Computer Security Architecture Workshop (CSAW)*, Fairfax, VA, November 2007.

[2] O. Acıiçmez, J. Seifert, and C. Koc. Micro-architectural cryptanalysis. *IEEE Security and Privacy Magazine*, 5(4), July-August 2007.

[3] A. Akavia, S. Goldwasser, and V. Vaikuntanathan. Simultaneous hardcore bits and cryptography against memory attacks. In *TCC*, pages 474–495, 2009.

[4] D. J. Bernstein. Cache-timing attacks on AES. `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`, Apr. 2005. Revised version of earlier 2004-11 version.

[5] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. Mukherjee, and R. Rangan. Computing architectural vulnerability factors for address-based structures. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 532–543. IEEE Computer Society, 2005.

[6] L. Bolotnyy and G. Robins. Physically unclonable function-based security and privacy in RFID systems. In *Pervasive Computing and Communications, 2007. PerCom'07. Fifth Annual IEEE International Conference on*, pages 211–220. IEEE, 2007.

[7] V. Boyko. On the security properties of oaep as an all-or-nothing transform. In *CRYPTO*, pages 503–518, 1999.

[8] R. Canetti, Y. Dodis, S. Halevi, E. Kushilevitz, and A. Sahai. Exposure-resilient functions and all-or-nothing transforms. In *EUROCRYPT*, pages 453–469, 2000.

[9] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Ann. Math. Statistics*, 23:493–507, 1952.

[10] A. Corporation. White paper: Understanding actel antifuse device security, January 2004.

[11] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. In *34th Intl. Symposium on Computer Architecture (ISCA)*, 2007.

[12] F. Davi, S. Dziembowski, and D. Venturi. Leakage-resilient storage. In *International Conference on Security and Cryptography for Networks (SCN '10)*, volume 6280 of *Lecture Notes in Computer Science*, pages 121–137. Springer, 2010.

[13] S. Dziembowski and K. Pietrzak. Leakage-resilient cryptography. In *FOCS*, pages 293–302, 2008.

[14] K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic Analysis: Concrete Results. In *Cryptographic Hardware and Embedded Systems*, volume 2162 of *Lecture Notes in Computer Science*, pages 251–261. Springer-Verlag, 2001.

[15] B. Gassend. *Physical random functions*. PhD thesis, Citeseer, 2003.

[16] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and hash trees for efficient memory integrity verification. In *In 9th Intl. Symp. on High Performance Computer Architecture*, pages 295–306, 2003.

[17] S. Haddad, C. Chang, B. Swaminathan, and J. Lien. Degradations due to hole trapping in flash memory cells. *IEEE Electron Device Letters*, 10(3):117–119, Mar. 1989.

[18] A. Huang. *Hacking The Xbox: An Introduction to Reverse Engineering*. No Starch Press, 2010.

[19] Y. Ishai, A. Sahai, and D. Wagner. Private circuits: Securing hardware against probing attacks. In *CRYPTO*, pages 463–481, 2003.

[20] U. R. Karpuzcu, B. Greskamp, and J. Torrellas. The bubblewrap many-core: popping cores for sequential acceleration. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 447–458, New York, NY, USA, 2009. ACM.

[21] J. Kelsey, B. Schneier, C. Hall, and D. Wagner. Side channel cryptanalysis of product ciphers. *Journal of Computer Security*, 8(2–3):141–158, 2000.

[22] P. Kocher, J. J. E, and B. Jun. Differential power analysis. In *Advances in Cryptology*, pages 388–397. Springer-Verlag, 1999.

[23] A. Kolodny, S. Nieh, B. Eitan, and J. Shappir. Analysis and modeling of floating-gate eeprom cells. *Electron Devices, IEEE Transactions on*, 33(6):835 – 844, June 1986.

[24] R. B. Lee, P. C. S. Kwan, J. P. Mcgregor, J. Dwoskin, and Z. Wang. Architecture for protecting critical secrets in microprocessors. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, 2005.

[25] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. *SIGPLAN Not.*, 35:168–177, November 2000.

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

[27] S. Micali and L. Reyzin. Physically observable cryptography. In *TCC 2004, LNCS*, pages 278–296. Springer, 2003.

[28] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *36th Annual International Symposium on Microarchitecture (MICRO)*, pages 29–40, December 2003.

[29] D. Page. Partitioned cache architecture as a side channel defence mechanism. In *Cryptography ePrint Archive, Report 2005/280*, August 2005.

[30] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.

[31] R. L. Rivest. All-or-nothing encryption and the package transform. In *FSE*, pages 210–218, 1997.

[32] O. Ruwase, P. B. Gibbons, T. C. Mowry, V. Ramachandran, S. Chen, M. Kozuch, and M. Ryan. Parallelizing dynamic information flow tracking. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 35–45, New York, NY, USA, 2008. ACM.

[33] A.-R. Sadeghi and D. Naccache, editors. *Towards Hardware-Intrinsic Security*. Springer, 2010.

[34] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[35] M. Shatzkes and Y. Huang. Characteristic length and time in electromigration. *Journal of Applied Physics*, 74(11):6609 –6614, Dec. 1993.

[36] S. W. Smith and S. Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31(8):831 – 860, 1999.

[37] G. Suh and S. Devadas. Physical unclonable functions for device authentication and secret key generation. In *Design Automation Conference, 2007. DAC'07. 44th ACM/IEEE*, pages 9–14. IEEE, 2007.

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

[39] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 85–96, New York, NY, USA, 2004. ACM Press.

[40] A. Tiwari and J. Torrellas. Facelift: Hiding and slowing down aging in multicores. In *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, pages 129 –140, nov. 2008.

[41] M. Tiwari, J. K. Oberg, X. Li, J. Valamehr, T. Levin, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood. Crafting a usable microkernel, processor, and i/o system with strict and provable information flow security. In *International Symposium of Computer Architecture (ISCA)*, 2011.

[42] Topham and Gonzalez. Randomized cache placement for eliminating conflicts. *IEEETC: IEEE Transactions on Computers*, 48, 1999.

[43] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. Rifle: An architectural framework for user-centric information-flow security. In *the 37th IEEE/ACM International Symposium on Microarchitecture*, pages 243–254. IEEE Computer Society, 2004.

[44] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. FlexiTaint: A programmable accelerator for dynamic taint propagation. In *Fourteenth International Symposium on High Performance Computer Architecture (HPCA)*, pages 196–206, New York, NY, USA, 2008. ACM.

[45] Z. Wang and R. Lee. New cache designs for thwarting cache-based side channel attacks. In *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, CA, June 2007.

[46] Z. Wang and R. Lee. A novel cache architecture with enhanced performance and security. In *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, pages 83 –93, nov. 2008.

[47] T. Wollinger and C. Paar. *New Algorithms, Architectures and Applications for Reconfigurable Computing*, chapter Security Aspects of FPGAs in Cryptographic Applications, pages 265–278. Springer, Cambridge, MA, 2005.

[48] L. Yang and L. Peng. Seccmp: A secure chip-multiprocessor architecture. In *Proceedings of the First Workshop on Architectural and System Support for Improving Software Dependability (ASID'06)*, San Jose, CA, October 2006.

[49] M. Zhang and N. Shanbhag. Soft-error-rate-analysis (sera) methodology. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(10):2140–2155, 2006.