Faculty of Engineering

Cybersecurity Lab

Implementation and Side-Channel Analysis of a Redundancy Masking Scheme and a Consolidated Linear Masking Scheme for AES

Yair Oren

Dan Michaeli

Submitted as requirement for a 4th year project for a bachelor's degree in engineering

Academic Supervisors: Prof Itamar Levi and Prof Osnat Keren

October 2024

Contents

List of Figures

1 Abstract

Masking is the most common implementational countermeasure against power analysis attacks. Current masking schemes tend to be expensive and cause high area and randomness overheads. In an attempt to fix this, masking schemes were proposed tailored specifically to the common AES cipher. They have lower cost and their protection order is more granular than of classical masking schemes. The first, RAMBAM, passes values through an isomorphism L then randomizes them with multiples up to order d of the polynomial P generating the finite field creating the isomorphism; field operations are done instead in a ring R_{PQ} that contains the isomorphic field representation. The second, CLM, adds additional low-cost randomness to RAMBAM by randomizing P and L per encryion cycle; it performs operations via a form of refresh arithemetic replacing reduction modulo PQ, thus eliminating the reliance on Q.

Both masking schemes are implemented, and their cost is shown to highly depend on d, and in the case of RAMBAM, on P and Q. RAMBAM in its suggested implementation is shown to be weak to side-channel attacks at large trace counts, with particular fixed values more vulnerable than others due to its underuse of randomness refresh in the multiplication step. Our results show that despite that the scheme offers partial protection even when $d/8$ is lower than the order of the attack and when only a partial amount of randomness is used.

2 Introduction

Side-channel attacks are a class of cryptologic attacks that exploit information leakage from the physical implementation of cryptographic algorithms. Deviations in sources like power consumption, electromagnetic emissions and timing information measured on a cryptographic device during its operation are used to reveal sensitive or secret data, such as secret keys. Various attacks exploiting this leakage, such as Differential Power Analysis [1] and Correlational Power Analysis [2], and various countermeasures against these attacks have been suggested in recent years.

A power measurement as a function of time of a cryptographic device encrypting some plaintext with some key is called a power trace. In the general setting of a power analysis attack, an encryption device with a known encryption algorithm and an unknown constant key k is given, and the attacker can extract power traces from the device with plaintexts known to him. The attacker's goal is to extract the encryption key using the power traces extracted from the device. One important prerequisite for an attack to succeed is that any power trace leaks information about the internal values processed in the encryption device.

The main goal of an attack countermeasure is to reduce the information power traces leak about the algorithmic intermediate values. Many such countermeasures exist, and they can be roughly divided into 2 main categories. The first type of countermeasure aims to reduce the signal-to-noise ratio (SNR) in the traces, by either reducing the power consumption of the cryptographic element itself or increasing the independent noise in the system. The earliest-researched countermeasures, such as [3], are of this type, but more recent work has been done in this field as well [4, 5]. The second type of countermeasure aims to make the power consumption distribution at each point of time independent of the internal algorithmic values, therefore invalidating attacks relying on statistical differences. This category of countermeasures includes dual rail logic, suggested in [6] and further developed in [7, 8], in which every value processed in the system, in the logic-gate level, has its boolean inverse also processed at the same time; and masking, discussed below, which is widely used in practice.

Masking schemes attempt to hide the algorithmic intermediate values calculated during the encryption by performing a random operation on them. By that, the power consumption and the real intermediate values are decorrelated. Several known masking schemes include the boolean masking scheme [9] and inner product masking [10]. Masking has several security proofs that verify its security against various types of attacks [9, 11]. Although effective against them, any type of masking necessitates the usage of masked

operations, which typically signifcantly increase the size, area and power of the circuit, and require large amounts of added randomness to be secure. Therefore, many masking schemes have been proposed that attempt to reduce the size and needed randomness of masked circuits while preserving security. In our work, we discuss 2 such masking schemes, tailored specifically to the popular AES encryption algorithm: RAMBAM, proposed in [12], and a suggested imporvement and generalization to it, CLM, proposed in [13]. We strive to physically verify and elaborate on the results simulated and suggested in the paper introducing CLM, and further elaborate on the considerations brought up in the paper introducing RAMBAM.

The work is structured as follows: subsection 3.1 provides details about the execution of side-channel attacks and security evaluation, subsection 3.3 provides prerequisites in coding theory and subsection 3.4 details the stages of the AES-128 cipher. Section 4 deals with the RAMBAM scheme, its details, implementation and implementation considerations, and finally its weaknesses and results of attacks on these weaknesses. Section 5 deals with the CLM scheme, its advantage as a generalization of RAMBAM, and its implementation. Finally, we draw conclusions from our results in section 6.

3 Background

3.1 Power Analysis Attacks & Security Evaluation

In the basic type of attacks, knowledge of the encryption algorithm allows partitioning the encryption key and plaintext into independent blocks of "managable" size, such that knowledge of some internal values may reveal the blocks' values. The point in time in which information about these specific internal values in leaked in the power trace is called the point of interest (POI).

The most basic power analysis attack is the differential power analysis attack (DPA) as given by Kocher in [1]. For this attack, some partitioning function $\ell\colon K \times P \to \{0,1\}$ – where K is the set of possible keys and for a small independent block and P is the set of plaintexts for that same block. A good partitioning function, which would result in a successful attack, should, when given a correct key, partition the plaintexts such that there is a statistical difference between the distribution of traces corresponding to the 2 sets of plaintexts; and when given an incorrect key, partition the plaintexts such that there is no such statistical difference. A common partitioning function is the value of some bit in a partial simulation of the encryption, e.g.

$$\ell(k,p) = (f(k \oplus p))_i$$

where f is a nonlinear function.

The attack proceeds as follows, where a superscript of B denotes the B'th block:

Algorithm 1 The DPA Attack

1: **procedure** DPA(L, T, ℓ, t_{thresh})
2: **Inputs:** $L_{N\times 1}$ – a list of plaintexts, $T_{N\times t_{max}}$ – a trace list, ℓ – a partitioning function, t_{thresh} – a threshold for the t-test (see below).
3: **Output:** $k_{guessed}$ – a key.
4: Initialize $k_{guessed} \leftarrow 0$
5: **for each** block B **do**
6: **for each** key guess $k \in K$ **do**
7: Partition $A \leftarrow \{i \in [N] \colon \ell(k, L_i^B) = 0\},\ B \leftarrow \{i \in [N] \colon \ell(k, L_i^B) = 1\}$
8: Partition the traces $T_A \leftarrow T_{A,:},\ T_B \leftarrow T_{B,:}$ ▷ Corresponding rows in T
9: **for** $t \leftarrow 0$ **to** t_{max} **do**
10: $tv \leftarrow t\text{-test}(C_t(T_A), C_t(T_B))$ ▷ Columns, represent a time slice
11: **if** $|tv| > t_{thresh}$ **then**
12: $k_{guessed}^B \leftarrow k$ ▷ t is the POI
13: **go to line** 5
14: **end if**
15: **end for**
16: **end for**
17: **end for**
18: **return** $k_{guessed}$
19: **end procedure**

In the security evaluation setting, an encryption device is given, with the evaluator free to capture power traces from for any arbitrary plaintexts and keys. The goal in this setting is to check whether the device is secure against power analysis attacks of all kinds – not just some specific kinds of attacks: this is done using tests that check whether there is any statistically significant leakage from power traces captured from the device.

Each test is performed very similarly to a DPA attack, but with a known key: some partitioning function ℓ and some key k are chosen, then traces are captured from the device with plaintexts corresponding to the different values of the partitioning function. The traces are then partitioned, and a t-test is computed between the 2 partitions. If the t-test returns a value greater than a specified threshold, the test fails and returns a result of there being leakage in the power traces. 2 main types of such tests exist:

1. In a fixed vs. fixed (FvF) test, plaintexts for each trace partition are chosen randomly from all plaintexts that yeild the corresponding value in the partitioning function.

2. In a fixed vs. random (FvR) test, for one trace partition the plaintexts are chosen randomly from all plaintexts that yeild some value in the partitioning function, and for the other trace partition the plaintexts are chosen entirely randomly. A

common class of FvR test simply fixes the plaintext in the "fixed" partition to be a certain fixed value.

Generally, when a fixed vs. random test gives a positive result, a matching fixed vs. fixed test will give a positive result, but with higher confidence. For both types of tests, any internal randomness in the encryption algorithm is chosen entirely randomly. The choice of the partition function is entirely up to the evaluator: there may be tests that yeild a positive result and tests that yeild negative results for the same trace sets.

Test vector leakage assessment (TVLA) is a standardized security evalutation methodology proposed by NIST [14]. It specifies what tests and how many of them should be run on the cryptographic device, how traces should be collected and what processing and preprocessing should be done to them, and how to and how to assess the security of the device from the results of all the tests. Most importantly, it specifies that in FvF and FvR tests the traces from the 2 partitions should be randomly interleaved with each other, to eliminate DC bias.

3.1.1 The Student's t-Test

The statistical tool used in DPA, side channel evaluation with TVLA and some other more sophisticated attacks is the Student's t-test. It is a statistical test which accepts 2 sets of measurements and gives measure of confidence for the hypothesis that the 2 sets are sampled from distributions with different means. Denoting the set on which the statistic is computed by a subscript, the t-score and the degree of freedom v are computed as follows:

$$t = \frac{\mu_0 - \mu_1}{\sqrt{\frac{\sigma_0^2}{n_1} + \frac{\sigma_1^2}{n_1}}}, v = \frac{\left(\frac{\sigma_0^2}{n_0} + \frac{\sigma_1^2}{n_1}\right)^2}{\frac{\left(\frac{\sigma_0^2}{n_0}\right)^2}{n_0 - 1} + \frac{\left(\frac{\sigma_1^2}{n_1}\right)^2}{n_1 - 1}}$$

the measure of confidence p is computed from t and v using the Student's t-distribution $P(t; v)$. Generally, a t-score of absolute value 4.5 or more is taken as a positive result in the test. As shown by Schneider & Morady [15], the t-test can be generalized to higher order statistical moments, to distinguish between distributions with equal lower moments but s different higher moments, such as in the case of boolean masking. Algorithmically, this may be done using raw moment calculation, using a 1-pass algorithm, or by a preprocessing of the traces.

3.2 Power Analysis Countermeasures

As stated in the introduction, masking is a class of countermeasures in which every value prodessed in the system is split into shares, which are randomized functions of the original value that the original value can be extracted out of. This randomness is added to the algorithm as an additional input, and may be extracted on the cryptographic device itself or sourced externally. Masking may be implemented either on the gate level, meaning every gate on the system is masked, or on the algorithmic level, where a masking scheme is tailored to the encryption algorithm itself. The simplest masking scheme is the order-d boolean masking scheme, where every bit b in the system is split into $d+1$ shares of a bit each, with values $(r_1, r_2, \ldots, r_d, r_{d+1})$ satisfying $b = \bigoplus r_i$ and d out of the $d+1$ shares being uniform random bits.

Security evaluation of any masking scheme may be done using 2 leakage models. In the first (the probing model, introduced in [16]) it is assumed that the attacker has N independent probes, each capable of measuring exactly the value of a single bit in the system as a function of time. A secure masking scheme in this model is one in which the attacker cannot extract with high probability the internal algorithmic value using the probes' measurements. While this model is useful in constructions of masking schemes, it does not reflect the abilities of attackers in physical systems, in which the power measurements from each probe will contain interference from other computations performed in the system. In the second model (the noisy leakage model, proposed in [9]), it is assumed the attacker has access to the hamming weight of the entire value processed in the system at any given time, disrupted by some additive Gaussian noise. In this model, a secure masking scheme is one in which the attacker cannot, given a set of these noisy measurements and using statistical testing like the t-test, distinguish between the leakage distributions corresponding to different algorithmic values. A parallel to the probing model with N probes in this model is one in which the attacker may perform statistical tests only pertaining to the Nth-order or below moments of the leakage distribution.

The order-d boolean masking scheme described above, for example, is secure in both models for $N \leq d$, but insecure for $N > d$. In the probing model, if the attacker has probes on all shares except r_i, he may only compute $b \oplus r_i$, which distributes uniformly; and given more probes he may simply calculate b by XORing all the probe readings. In the noisy leakage model, the leakage distributions conditional on $b = 0$ and on $b = 1$ are mixtures of Gaussians with equal moments up to the dth order but different in the $(d+1)$th order.

Using any masking scheme necessitates adapting all computations in the cryptographic

device to work on masked data, using masked gates or masked operations. Typically, when using a masking scheme secure up to the dth order, adapting any linear (XOR or NOT) operation requires an $O(d)$ inflation in the size of the circuit, while adapting any nonlinear operation requires an $O(d^2)$ inflation in the size of the circuit. For example, again taking the simple boolean masking and given 2 masked bits represented by vectors $\vec{a}$ and $\vec{b}$, implementing a XOR gate requires a bitwise XOR between the 2 vectors, and implementing an AND gate requires computing the Kronecker product of the 2 vectors then XORing the rows. Specifically for $d = 1$ this becomes

$$\begin{pmatrix} r_1 \\ a \oplus r_1 \end{pmatrix} \oplus \begin{pmatrix} r_2 \\ b \oplus r_2 \end{pmatrix} = \begin{pmatrix} r_1 \oplus r_2 \\ a \oplus b \oplus r_1 \oplus r_2 \end{pmatrix}$$

and

$$\begin{pmatrix} r_1 \\ a \oplus r_1 \end{pmatrix} \otimes \begin{pmatrix} r_2 \\ b \oplus r_2 \end{pmatrix} = \begin{pmatrix} r_1 r_2 & r_1 (b \oplus r_2) \\ r_2 (a \oplus r_1) & (a \oplus r_1)(b \oplus r_2) \end{pmatrix}$$

XORing the rows gives

$$\begin{pmatrix} r_1 b \\ (a \oplus r_1) b \end{pmatrix}$$

A problem arising in masked gates and operations is the need to "refresh" the randomness by adding new random bits after any non-linear operation, since the non-linearity of the operation may leak information about the original masked datum. Taking the example of the masked AND gate above, the top share in the computed product leaks information on b: if the top share is equal to 1, the attacker immediately knows that $b = 1$. Therefore, before XORing the rows is done, it is necessary to XOR some random bit r_3 to both shares of the result. The refresh may in some cases neccessitate an addition of $O(d^2)$ random bits per nonlinear operation.

3.3 Finite Fields & Error Correcting Codes

Given an irreducible polynomial $\pi(x)$ with coefficients in GF(p), the set of polynomials over GF(p) with addition and multiplication modulo $\pi(x)$ is a finite field. If $\deg \pi = d$, then the field has p^d elements and is denoted by GF(p^d) or by F_π. Constructions of a finite field from different polynomials of the same degree are isomorphic as fields, meaning there is a mapping from one construction to another that preserves addition and multiplication.

The element corresponding to the polynomial $x \mod \pi$ is denoted by α. Every finite field GF(p^d) has an element of order $p^d - 1$, which is called a primitive element. In fact,

there are $\phi(p^d - 1)$ of those, where ϕ is Euler's totient function. An isomorphism can be built between $F_{\pi_1}$and F_{π_2} by taking $\beta \in F_{\pi_1}, \gamma \in F_{\pi_2}$ both primitive elements with the same minimal polynomial over $\mathrm{GF}(p)$, then mapping $\beta \mapsto \gamma$. Completion of the isomorphism is done from its multiplicity.

For every finite field $\mathrm{GF}(p^d)$ generated from $\mathrm{GF}(p)$ (said to be of *characteristic* p) it holds that for every $m \in \mathbb{N}$ and $x, y \in \mathrm{GF}(p^d)$

$$(x + y)^{p^m} = x^{p^m} + y^{p^m}$$

(the "freshman's dream" theorem).

A linear error correcting code of length n and dimension d over $\mathrm{GF}(q)$ is a k-dimensional subspace of the vector space $\mathrm{GF}(q)^n$. The minimal distance of a code C is defined as $d = \min_{c_1, c_2 \in C} \mathrm{HD}(c_1, c_2)$. In a linear code, this simplifies to $d = \min_{c \in C, c \neq 0} \mathrm{HW}(c)$. An encoding is an invertible function that maps information words, which are vectors in $\mathrm{GF}(q)^k$, to codewords, which are vectors in C. A systematic encoding is one in which the information word appears intact at the start or end of the codeword. The transformation matrix of an encoding is called a generator matrix for the code.

For every linear error-correcting C code there exist matrices H of dimension $(n - k) \times n$ which satisfy that for every $c \in \mathrm{GF}(q)^n$, $c \in C$ if and only if $cH^T = 0$. Any matrix satisfying this property is called a parity-check matrix for the code, and the vector yH^T for any $y \in C$ is called the syndrome of y. Given a systematic generator matrix of the form $G = (B \mid \mathrm{I}_{k \times k})$ there exists a parity-check matrix for the same code of the form $H = \left(\mathrm{I}_{n-k \times n-k} \mid -B^T\right)$.

A cyclic error-correcting code of length n over $\mathrm{GF}(q)$ is a principal ideal $(g) \subset {}^{\mathrm{GF}(q)[x]}\!/_{(x^n-1)}$. The polynomial $g(x) \mid x^n - 1$ is called the generator polynomial of the code, and it is the lowest-degree codeword up to multiples by a constant. Its degree r satisfies $r = n - k$, where k is the dimension of the cyclic code, viewed as a linear code. A non-systematic encoding for a cyclic code is the convolution encoding $m(x) \mapsto m(x)g(x)$. A systematic encoding for a cyclic code is the mapping $m(x) \mapsto m(x)x^r - (m(x)x^r \mod g(x))$.

A shortened cyclic code is a linear, non-cyclic code obtained by first taking some $[n, k]$ cyclic code then selecting from it only the codewords matching information words of degree $< k'$ in the convolution encoding. Its length and dimension are $n - (k - k')$ and k', respectively.

Important Note About Conventions Polynomials, and by extension field elements, can be viewed as row vectors with length equaling the degree of the polynomial plus one. Thoroughout, we will be using a little-endian convention, where the leftmost or uppermost symbol in the vector corresponds to the lowest degree coefficient in the polynomial.

3.4 The Advanced Encryption Standard (AES)

The AES-128 (Rijndael) cipher [17] is a block cipher that uses an architecture of a substitution-permutation network (SPN) with 10 rounds. The plaintext, ciphertext and key lengths are 128 bits each. The algorithm's intermediate values are viewed as 4 by 4 column-major order matrices of byte-size elements, called state vectors. Each byte is viewed as an element of F_{P_0}, where

$$P_0(x) = 1 + x + x^3 + x^4 + x^8$$

is an irreducible poylnomial over GF(2).

Each round consists of the following steps, in order:

AddRoundKey: The state vector is XORed with the current round key.

SubBytes (aka S-Box): Each byte of the stte vector is passed through a nonlinear function. Viewed as elements of the field F_{P_0}, the function can be first viewed as first inversion in the field (equivalent to raising to the power of 254), then passing through an affine transformation $x \mapsto Wx + w$, where

$$W = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}, w = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

ShiftRows: Each row in the state vector is shifted by a certain amount, with the i'th row of the state vector being shifted i elements to the left.

MixCols: A field-linear operation is applied over each of the column of the state vector:

each column is multiplied by the matrix

$$M = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix}$$

of elements in F_{P_0}, where the elements 1, 2 and 3 correspond to the field elements 1, α and $1 + \alpha$, respectively.

In the final (10th) round, the MixCols step is skipped. After the final round, an additional AddRoundKey step is performed.

3.4.1 The Key Schedule Sub-Algorithm

A different key is used for each of the 10 rounds of the encryption scheme, and an additional one is used for the AddRoundKey step after the final round. The key for each round is obtained from the key of the previous round using a process called a key schedule (or a key-expansion).

A word is defined as a 4 by 1 vector of byte-size elements. For each round $1 \leq i \leq 10$, define the word $rcon_i$ as $rcon_i = \begin{pmatrix} \alpha^{i-1} & 0 & 0 & 0 \end{pmatrix}$. Define the RotWord operation as a left shift of a word, and the SubWord operation as an element-wise application of the SubBytes step to a word.

Denoting the words of the original key as K_i, where $0 \leq i \leq 3$, and the words of the round keys as W_i, where $0 \leq i \leq 43$, the key-expansion algorithm works as follows:

$$W_i = \begin{cases} K_i & i < 4 \\ W_{i-4} \oplus \text{SubWord}(\text{RotWord}(W_{i-1})) \oplus rcon_{i/4} & i \geq 4, i \equiv 0 \pmod 4 \\ W_{i-4} \oplus W_{i-1} & else \end{cases}$$

4 Previous Work – the RAMBAM Scheme

n their 2022 paper, Belenky et al. [12] suggested a masking scheme specifically tailored for AES that is called Redundancy AES Masking Basis for Attack Mitigation, or RAMBAM. It is a form of algebraic masking, in which rather than splitting the computed data into shares, some form of random algebraic manipulation is done onto it. The discussion in sections 4.1 and 4.2 is an abridged form of the discussion in the paper.

4.1 Details

Throughout the discussion we will be using the notation $m = 8$.

First, note that though the polynomial P_0 in the AES algorithm was chosen for its cryptoanalytic strength [17], it is not unique, in that any other irreducible polynomial P can be taken as a generating poylnomial for the field $\mathrm{GF}(2^m)$. There is isomorphism $L\colon F_{P_0} \to F_P$, and since all steps in the AES cipher, besides the affine transformation in the S-Box, are simple field operations, they commute with L. Thus, an equivalent cipher can be made by first transforming all inputs to the cipher by L, then applying the steps in the original cipher – replacing *rcon* and the MixCols matrix with element-wise applications of L on them – and finally transforming all outputs back to F_{P_0} using the inverse L^{-1}. For a complete discussion on how to transform the affine transformation to the field F_P, see section 4.1.1.

After applying this transformatoin, additinal redundancy bits can be added by choosing any polynomial Q of degree d, and extending all operations in the cipher to the ring R_{PQ} of all polynomials modulo PQ. In this ring, operands are represented by $(m + d)$-bit vectors. There exists a ring homomorphism $H\colon R_{PQ} \to F_P$, which has the simple form of $H(f(x)) = f(x) \mod P$. No transformation of the inputs is needed, and again since all cipher operations are field operations, they commute with H and therefore a simple application of H in the output step is sufficient to return to the original cipher in F_P.

Randomness for masking is then incorporated by adding random multiples of P to intermediate values in the cipher: first to the every byte of the input, then as refresh after before any data reuse. The multiples are by polynomials of degree $< d$, of which 23 are needed – 16 for input randomization and 7 for refreshes in the S-Box, which are reused across applications of the S-Box. The S-Box is split into parts, with a refresh after each one. An equivalent perspective to this is that the unmasked data are syndromes, and masking is done using random codewords of the shortened error-correcting code of length $m + d$ generated by P.

Every RAMBAM variant has predetermined d, P and Q. The choice of P and Q can impact the security of the masking scheme: for a discussion on this, see section 4.1.2.

4.1.1 Note On Picking the Affine Transformation

This discussion is taken almost exactly from [13].

The affine transformation used in the S-Box of any RAMBAM instance should commute with with the homomorphisms L and R: denoting the extended affine transformation matrix and vector by T and t respectively, it should hold that for every v reachable by the algorithm,

$$L^{-1}(H(T(v)+t)) = W(L^{-1}(H(v))) + w$$

It is useful to treat v as obtained via a systematic encoding, given from a vector x in F_{P_0} and a (random) length-d vector r via

$$v = (x \mid r)\, M$$

where M is the block matrix

$$M = \begin{pmatrix} L & 0_{m\times d} \\ B & I_{d\times d} \end{pmatrix}$$

and B being part of the systematic generator matrix for the shortened cyclic code: $G = (B \mid I_{m\times m})$. H then has the matrix form $H = \left(I_{d\times d} \,\middle|\, B^T\right)$ It then holds that $x = vH^TL^{-1}$. Plugging this into the condition above gives

$$xW + w = (vT + t)H^TL^{-1} = (x, r)MTH^TL^{-1} + tH^TL^{-1}$$

Taking $v = 0$ gives the condition on t

$$w = tH^TL^{-1}$$

or $wL = tH^T$ – that is, t is a codeword whose syndrome is tL. For the condition on T, it is useful to write T as a block matrix

$$T = \begin{pmatrix} T_{1,1} & 0_{m\times d} \\ T_{2,1} & T_{2,2} \end{pmatrix}$$

we can then expand

$$xW \stackrel{!}{=} (x,r)MTH^TL^{-1} = (x,r)\begin{pmatrix} LT_{1,1} & 0 \\ BT_{1,1}+T_{2,1} & T_{2,2} \end{pmatrix}\begin{pmatrix} I \\ B \end{pmatrix}L^{-1}$$
$$= (x,r)\begin{pmatrix} LT_{1,1} \\ BT_{1,1}+T_{2,1}+BT_{22} \end{pmatrix}L^{-1}$$

since this expression needs to hold for every x and r, we must have $W = LT_{1,1}$ and $BT_{1,1} + T_{2,1} + BT_{22} = 0$. The first equation determines $T_{1,1}$: $T_{1,1} = L^{-1}WL$. The second gives a condition on $T_{2,1}$ and $T_{2,2}$. It is useful to simply take $T_{2,2} = I_{d\times d}$, and then get the condition $T_{2,1} = B(L^{-1}WL + I_{m\times m})$. This completely determines the matrix T.

4.1.2 Note on Picking P and Q

We would wish to choose polynomials for P and Q so that the security is maximized. There are several considerations to take into account.

Mathematical Constraints First, note that P must be irreducible while Q doesn't, since P is used to create a finite field F_P while Q is only used to create a ring R_{PQ} which doesn't require Q to be irreducible, as mentioned above.

Maximizing Security The main idea in RAMBAM is that each polynomial in the field F_P have multiple redundant representations in R_{PQ}, and consequently, the power consumed for different intermediate values will be about the same in average. This is called uniformity and we wish to maxamize it. We present several ideas for P and Q for this purpose, most of them are taken from [12].

Q not Divisible by P Assume Q is divisible by P - $Q(x) = M(x)P(x)$. Since R_{PQ} is a ring of charcaristic 2 then $(a+b)^{2^m} = a^{2^m}+b^{2^m}$ for each $a, b \in R_{PQ}, n \in \mathbb{N}$. Now, note that $(T(x) + r(x)P(x))^{2^m} \mod PQ = T^{2^m}+r^{2^m}P^{2^m} \mod P^2M = T^{2^m}+r^{2^m}P^{2^m} \mod P^2M$ and let $r = kM + m$ be the decomposition of r modulo M. Then $(T(x) + r(x)P(x))^{2^m} \mod PQ = T^{2^m} + m^{2^m}P^{2^m} + k^{2^m}P^{2^m-2}M^{2^m-1} \cdot P^2M \mod P^2M = T^{2^m} + m^{2^m}P^{2^m} \mod P^2M$, this means that in this case the result of raising to a power of 2 depends on a polynomial of degree $d - 8$ - m, instead of polynomial of degree d - r. As a result, the number of different results decreases by a factor of at least $2^8 = 256$ which may compromise the security of the components that raise to the power of some power of 2.

Q is Irreducible We now assume that Q is not divisible by P and P is irreducible as before. Thus, P and Q are relatively prime which means we can use the chinese remainder theorem to analyze polynomials modulo PQ. According to the theorem, instead of calculating the remainder $p(x) \mod PQ$, we can calculate $p(x) \mod P$ and $p(x) \mod Q$ separately and multiply them. Let X be element in F_P and let $X+r_1P$ be some redundant representation of X. In order to calculate $X + r_1P \mod PQ$ we can calculate $X + r_1P \mod Q$ and $X + r_1P \mod P$ instead. The remainder modulo P is just the original element in the field F_p. On the other hand, since P is invertible modulo Q, the redundant representations cover all 2^d possible values for the remainder modulo Q. Now, in order to multiply two elements in R_{PQ} all we need to do is to look at their corresponding remainders and multiply them. The multiplication modulo P is just the multiplying two elements in the field F_p. The multiplication modulo Q is essentialy multiplying all polynomials up to degree d by each other modulo Q - 2^{2d} multipliers. If Q is irreducible we will get 0 only when 0 is one of the multiplicands - $2^{d+1} - 1$ times - since there will be no zero divisors, while if it is reducible we will recieve 0 in each time that all of Q factors appear in the polynomials. Thus, in order to maintain the uniformity of the multiplier we will prefer irreducible Q.

Different Angle - Average Hamming Weight We know that the power consumption a typical device correlates strongly to hamming weight of the data processed in it. Thus, we will expect a "secure" component to have output hamming weight distribution centered at $0.5X$ when X is the largest possible hamming weight and small variance. That way, the power consumed for each calculation will be very similar and we won't be able to learn new information about the intermediate values from the power traces. In their work, [12] evaluated the security for different choices of P and Q using the TVLA methodology, and wrote the best and worst choices for P and Q security wise. Our assumption that components using the best choices will have a more similar distribution to those we described. For example, we simulated the operation of component that raises to the power of 2 over all possible inputs for the polynomials presented at [12]. These are the polynomials and their corresponding mean and variance - when the left side is the worst and the right side is the best security wise.

Redundancy	P	Q	mean	variance	P	Q	mean	variance
3	0x1dd	0xd	5.5000	3.1265	0x169	0x9	5.1250	2.6106
4	0x163	0x1f	5.8750	2.8601	0x163	0x17	5.6250	2.8601
5	0x1dd	0x33	6.5000	3.2504	0x1a9	0x3b	6.0312	3.3432
6	0x1f9	0x45	6.9688	3.4055	0x11b	0x47	6.6250	3.3596
7	0x1f5	0xff	7.5156	3.8592	0x187	0xfb	7.4141	4.0865
8	0x1a3	0x101	8.0000	6.0001	0x169	0x17b	7.5078	4.1953

As we expected, the best polynomials consistently gave us mean closer to $0.5(d+7)$ and most of the time gave us smaller variance, especially for $d = 8$.

4.2 Full Algorithm

The RAMBAM encryption algorithm, as appears in [12], is

Algorithm 2 RAMBAM(P, d, Q)

Require: P is an irreducible polynomial of degree 8, Q is a polynomial of degree d

function RAMBAM(x_{in}, k_{in}, r)
  Inputs: x_{in} – 16-byte plaintext, k_{in} – 16-byte key, r – 23 random d-bit values.
  Output: x_{out} – 16-byte ciphertext.
  for $i \leftarrow 0$ **to** 15 **do**
    $k_i \leftarrow L(k_{in,i})$
    $x_i \leftarrow L(x_{in,i}) + r_i P$
  end for
  for round $\leftarrow 0$ **to** 9 **do**
    $x \leftarrow$ ADDROUNDKEY(x, k)
    $k \leftarrow$ PROTECTEDNEXTROUNDKEY(k)
    $x \leftarrow$ PROTECTEDSUBBYTES(x, r)
    $x \leftarrow$ SHIFTROWS(x)
    if round $\neq 9$ **then**
      $x \leftarrow$ PROTECTEDMIXCOLUMNS(x)
    end if
  end for
  $x \leftarrow$ ADDROUNDKEY(x, k)
  for $i \leftarrow 0$ **to** 15 **do**
    $x_{out,i} \leftarrow L^{-1}(x_i \mod P)$
  end for
  return x_{out}
end function

Algorithm 3 The RAMBAM(P, d, Q) S-Box

function RAMBAMSBOX(x_{in}, r) ▷ All algebraic operations are performed in R_{PQ}
  Inputs: x_{in} – $(8+d)$-bit value, r – 7 random d-bit values.
  Output: x_{out} – $(8+d)$-bit value.
  $t \leftarrow x_{in}$
  $t_2 \leftarrow t^2 + r_0 P$
  $t_3 \leftarrow t_2 \cdot t + r_1 P$
  $t_{12} \leftarrow (t_3)^4 + r_2 P$
  $t_{14} \leftarrow t_{12} \cdot t_2 + r_3 P$
  $t_{15} \leftarrow t_{12} \cdot t_3 + r_4 P$
  $t_{240} \leftarrow (t_{15})^{16} + r_5 P$
  $t_{254} \leftarrow t_{240} \cdot t_{14} + r_6 P$
  $x_{out} \leftarrow \text{RAFF}(t_{254})$
  return x_{out}
end function

4.3 Implemetation Details

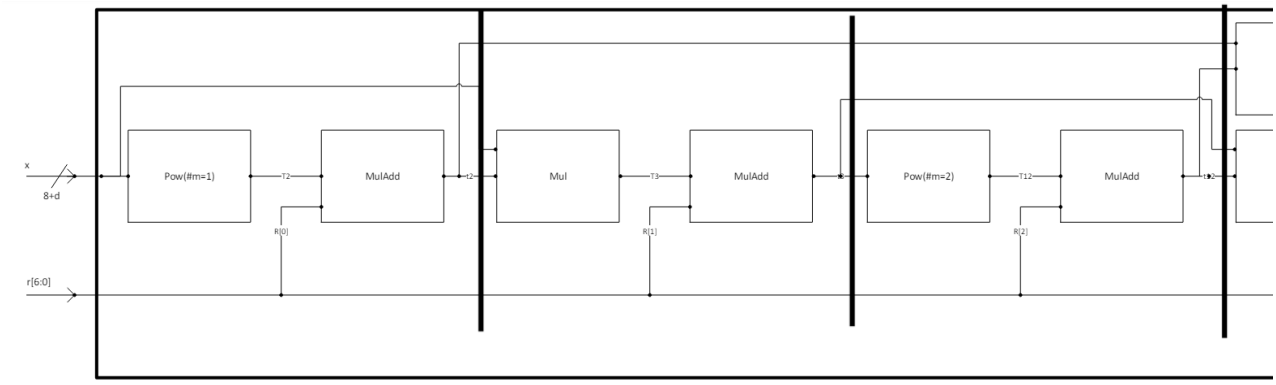
A sample implementation of RAMBAM by FortifyIQ was published recently in [18], but it uses hard-coded values for P, Q and d and so is of less use to us, particularly for transitioning to the work on CLM. Therefore, we implemented in SystemVerilog a full RAMBAM module, with the implementation details listed below. Whenever possible, we tried to use linear circuits (implementing matrix multiplication with constant matrices) rather than non-linear ones, since linear gates' outputs tend to leak less information about their inputs than non-linear ones'. All modules are parameterized by P, Q, d and derived parameters. Some of the matrices needed for the linear operations in the cipher are dependent on the security parameters in a non-trivial way; once they are generated, they are all fixed for a given RAMBAM variant, and so multiplication by them is a simple linear operation on the input bits. Note that the size of these matrices is quadratic with d, thus making the entire design's cost quadratic with d – similarly to the cost of masked operations in classical boolean masking.

4.3.1 The S-Box

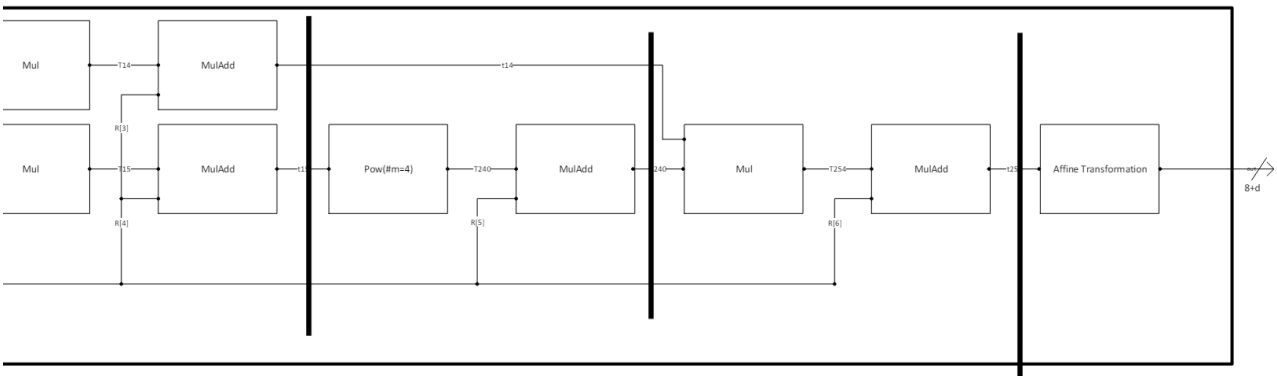
The RAMBAM S-Box takes as input an $m+d$-bit plaintext x and 7 random d-bit vectors $r_0, \ldots, r_6$, and outputs an $m+d$-bit ciphertext y.

For ease of measurement, the S-Box was partitioned into 7 stages, each taking a single clock cycle. The output of each cycle is stored in a register, which leaks measurable information. This division into clock cycles was done in order to both increase the

time the S-Box takes to operate and increase the measurement SNR – 2 considerations which would be deterimental in a typical implementation but are useful for clean trace collection. For a diagram of the S-Box, see figure 4.1



(a) First Part



(b) Second Part

Figure 4.1: Block diagram of the RAMBAM S-Box

The internal modules in the S-Box are listed below.

`power` The `power` module takes an $m+d$-bit input p_{in} and outputs an $m+d$-bit output p_{out}, both representing polynomials in R_{PQ}, and is additionally parameterized by an exponent e. Its output and input obey the relation

$$p_{out}(x) = (p_{in}(x))^{2^e}$$

Since raising to a power by a power of 2 is a linear operation in R_{PQ}, the module performs a linear operation and can be realized via matrix multiplication. The power operation maps the polynomial x^i to the polynomial x^{i2^e}, and so the rows

of the matrix are given by

$$R_i(M_e) = x^{i2^e} \mod PQ$$

`multiplier` The `multiplier` module takes 2 $m+d$-bit inputs p_1, p_2 and outputs an $m+d$-bit output p_{out}, all representing polynomials in R_{PQ}. Its output and input obey the relation

$$p_{out}(x) = p_1(x) \cdot p_2(x)$$

As suggested in the original paper, multiplication is realized using a series of add-and-shift operations, where the bits in p_2 control whether a shifted-and-reduced version of p_1 is added to an accumulator. Reduction in each step is done using a simple controlled XOR operation. We used a fully combinatorial design with no refresh between atomic operations; further elaboration on this can be found in section 4.3.4.

`mul_add` The `mul_add` module takes an $m+d$-bit input p_{in} and an d-bit input r, and outputs an $m+d$-bit output p_{out} – all representing polynomials in R_{PQ}. The output of the module is a randomized version of the input, given by encoding r then adding (using a simple XOR operation) the resultant codeword r_0P – with r_0 being related to r, but not neccessarily equal to it – to $p_{in}(x)$:

$$p_{out}(x) = p_{in}(x) + r_0(x)P(x)$$

2 variants of this module are tested: in the first, r is encoded using a convolution encoder, and in the second, r is encoded using a systematic encoder. Both are realized by matrix multiplication by a $d \times (m+d)$ matrix, with the rows matrix for the convolution encoder being shifted versions of P,

$$R_i(M_{\text{conv}}) = 0^i \parallel P \parallel 0^{d-1-i}$$

and the matrix for the systematic encoder having the form

$$M_{\text{sys}} = (B \mid \mathrm{I}_{d\times d})$$

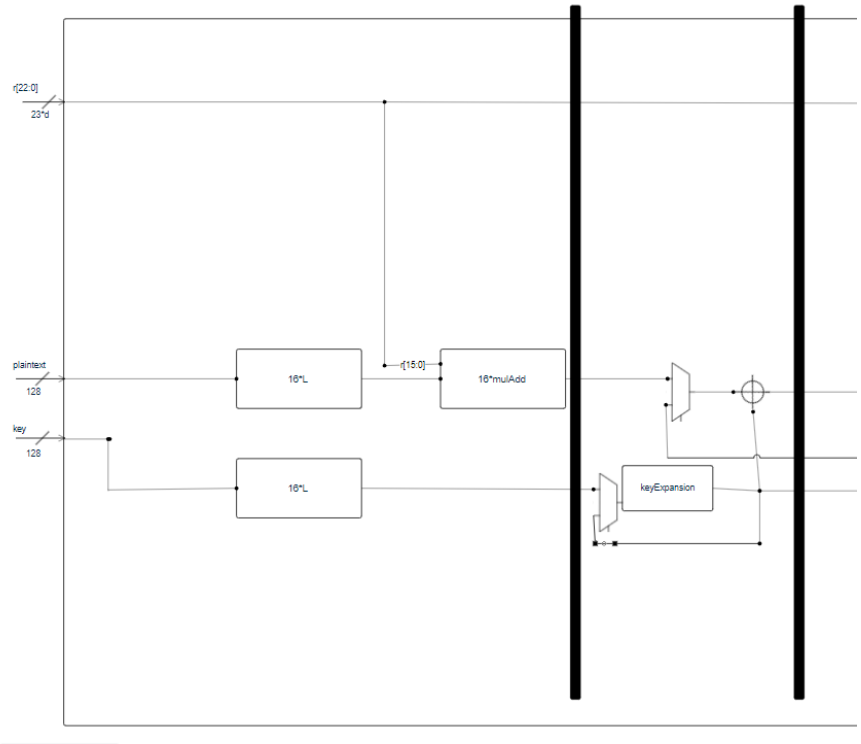
with the rows of B being given by

$$R_i(B) = x^{i+m} \mod P$$

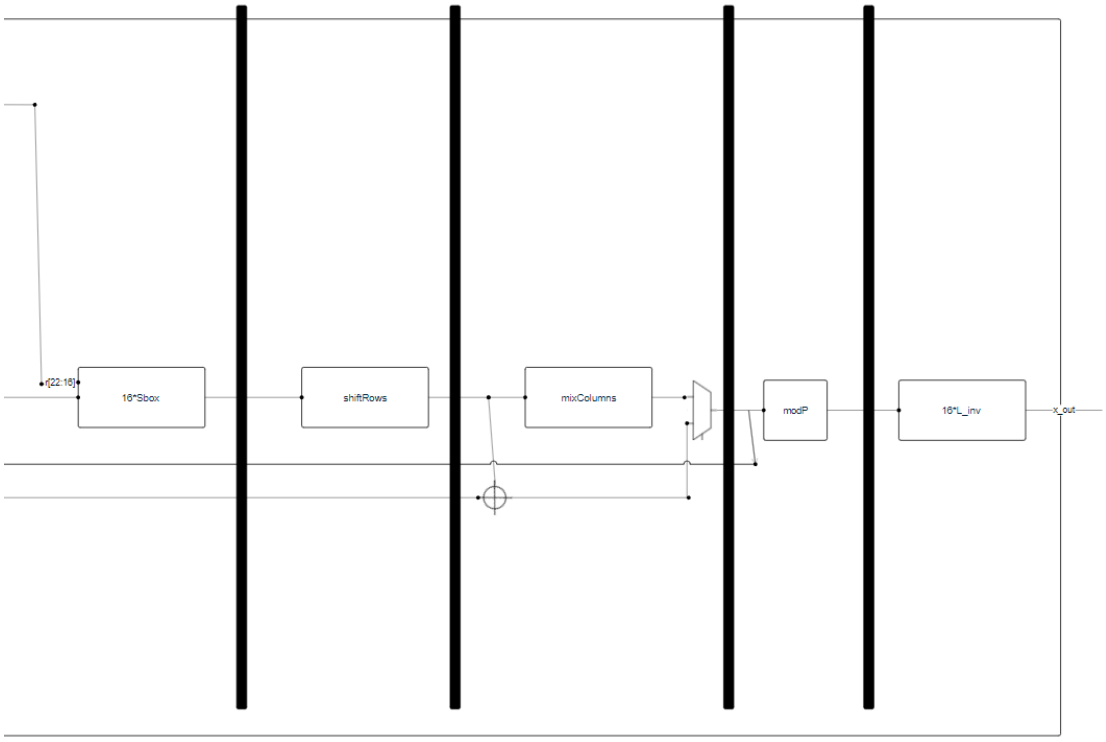
`affine_transform` The `affine_transform` module takes an $m + d$-bit input p_{in} and outputs an $m + d$-bit output p_{out}. The module performs an affine transformation on the input, with the matrix and the vector for the transformation being given in section 4.1.1.

4.3.2 Full AES Module

The full RAMBAM module takes as input 128-bit plaintext and key, and 23 random vectors of d bits, and outputs a 128-bit ciphertext. Additional `drdy` signals for external interfacing are also provided. The module uses a datapath and control paradigm: for the state machine governing the module's operation, see appendix A, and for the diagram of the datapath, see figure 4.2. An implementation with 16 S-Boxes working in parallel was chosen for ease of implementation: a sequential design with 1 S-Box would yield cleaner traces and be cheaper, but would take longer to operate.



(a) First Part



(b) Second Part

Figure 4.2: Datapath of the full RAMBAM module

Besides the S-Box, the RAMBAM module has several sub-modules, details for which are listed below.

`input_transform` & `output_transform` The `input_transform` and `output_transform` modules each take an m-bit input p_{in} and output an m-bit output p_{out}. Their functions are inverse – `input_transform` takes an input representing a polynomial in $F_{P_0}$and applies the isomorphism L to it to receive an output representing a polynomial in F_P, while `output_transform` takes an input representing a polynomial in F_P and applies the inverse of the isomorphism to receive an output representing a polynomial in F_{P_0}. Since both the isomorphism and its inverse are linear functions, both modules can be realized via matrix multiplication.

The matrices representing L and L^{-1} are both generated offline. The matrix representing L is found by using a brute-force approach to find the β, γ listed in section 3.3, then using the multiplicity of the isomorphism to calculate the rows of the matrix. The matrix representing L^{-1} is generated from the matrix representing L via Gaussian elimination.

`add_round_key` The `add_round_key` module takes as input 2 extended state vectors k and x, and outputs a state vector y. y is simply given by

$$y = x + k$$

2 instances of the `add_round_key` module are used: the first performs the AddRoundKey step before each round, and the second performs the AddRoundKey step – which replaces the MixCols step – after the final round. The first is fed k from a register storing the key state vector, while the other is fed k directly from the output of the `key_expansion` module, to cut on a clock cycle.

`shift_rows` The `shift_rows` module takes as input a state vector x and outputs a state vector y, with y being an application of the exact ShiftRows step from the AES cipher.

`mix_columns` The `mix_columns` module takes as input a state vector x and output a state vector y, with the output obtained from the input using an adapted MixCols step: denoting the $m+d$-bit words of any single column of x and the respective column in y by $b_0, \ldots, b_3$ and $f_0, \ldots, f_3$, respectively, and taking them as polynomials

in R_{PQ}, it holds that

$$\begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \end{bmatrix} = \begin{pmatrix} L(x) & L(1+x) & L(1) & L(1) \\ L(1) & L(x) & L(1+x) & L(1) \\ L(1) & L(1) & L(x) & L(1+x) \\ L(1+x) & L(1) & L(1) & L(x) \end{pmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Multiplications by $L(1)$, $L(x)$ and $L(1+x)$ are all linear in the multiplied element, so they can all be realized via matrix multiplications. Since L is an isomorphism, $L(1) = 1$ and $L(1+x) = 1 + L(x)$, and so the only nontrivial matrix that needs to be calculated is the matrix representation of multiplication by $L(x)$, M_{m_2}: the matrix representations of multipliying by $L(1)$ and $L(1+x)$ are given by $M_{m_1} = \mathrm{I}_{(m+d)\times(m+d)}$ and $M_{m_3} = \mathrm{I}_{(m+d)\times(m+d)} + M_{m_2}$, respectively. M_{m_2} is calculated offline: since multiplying by $L(x)$ maps the element x^i to the element $L(x)x^i$, the rows of M_{m_2} are given by

$$R_i(M_{m_2}) = \left(R_1(M_L)\right)(x) \cdot x^i \mod PQ$$

The module itself operates by first computing all the partial products in the matrix multiplication given above, then performing a XOR between them to get the outputs $f_0, \ldots, f_3$.

`key_expansion` The `key_expansion` module takes as input a state vector k_{in}, representing the key for the current round, and outputs a state vector k_{out}, equal to the key of the next round. Its function is very similar to the key expansion algorithm used in the original AES chiper, only now all S-Boxes are protected S-Boxes implemented exactly as in the previous section, and *rcon* passing through an `input_transform` module before being used. Therefore, this module receives as additional input a random vector of 7 d-bit random inputs to the S-Boxes. This module is a sequential module, and communicates with the main RAMBAM module via the inputs and outputs `drdy_i`, `drdy_o` and `first_round`. `first_round` is set by the main RAMBAM module on the first round, and tells the module to re-initialize *rcon* to 1: for the full state machine of this module see appendix A. We chose to use an implementation that generates a single round key each round, rather than one that precomputes and stores all the round keys. While the latter implementation is safer when there is significant key reuse (as in most practical applications), we implemented the first due to its ease of use, and delibaratly run tests on the plaintext and not the key of the cipher to avoid weaknesses stemming

from this.

`mod_P` The `mod_P` module takes as input an $m+d$-bit input p_{in}, representing a polynomial in R_{PQ}, and outputs an m-bit output p_{out}, representing a polynomial in F_P. The input and output obey the relation

$$p_{out}(x) = p_{in}(x) \mod P$$

since reduction modulo a fixed polynomial is a linear operation, this module can be realized via matrix multiplication – in fact, this matrix is a parity-check matrix of the shortened cyclic code generated by P. This matrix has the form

$$M_{\text{mod}} = \left(\frac{\mathrm{I}_{m \times m}}{B} \right)$$

with B given exactly as described above.

4.3.3 Alternatives for Parameter Extraction

As stated above, the matrices L, L^{-1}, M_{conv},M_{m_2}, B, M_1, M_2, M_4, $T_{1,1}$ and T_{22}, and the vector t, are all parameters of a RAMBAM instance that are dependent on P and Q. When designing the system, we were faced with a choice of whether to design hardware modules that accept the given implementation's P and Q and generate the matrices via logic operations or via a memory module, or to generate the matrices offline, via software, and load them to the design at compilation time. While a hardware-only approach is easier to generalize to the final CLM implementation, where P and Q are not given as parameters but rather as inputs (see section 5.1), there are 3 main advantages to using a mixed hardware-software design: the first is that some of the matrices have a dependency on P and Q that is hard to emulate in hardware; the second is that eliminating the matrix generation hardware modules saves on implementation area; and the third is that using matrices that are not fixed in synthesis time turns matrix multiplication into an operation needing non-linear gates, and this gates may leak internal data. These advantages led us to use the hardware-only approach: all parameters except M_{conv} are generated offline via a MATLAB script that accepts P and Q and outputs all the matrices to a SystemVerilog header.

4.3.4 Alternatives for Multiplier Implementation

The original RAMBAM paper gives the multiplier as pseudocode intended for software implementations. In our implementation, we wished to create a translation to hardware of this multiplier that is as faithful as possible, in order to capture any strengths and weaknesses present in the original software implementation. This implementation works as follows: denote

$$p_2(x) = \sum_{i=0}^{n} b_i x^i$$

then the result is

$$p_{out}(x) = \sum_{i=0}^{n} b_i \left(x^i p_1(x) \mod PQ \right)$$

The multiplication is performed as follows: an accumulator is set to 0 and an iteration loop with $m+d$ iterations is started. In each iteration, the corresponding b_i is multiplied (via an AND gate) with an $m+d$ bit shifter, which holds $x^i p_1(x) \mod PQ$, and added to the accumulator. The shifter is advanced from one iteration to the next by shifting it right – equivalent to multiplication by x – then reducing modulo PQ by adding PQ ANDed with the overflow bit from the shift (a module called `modular_shift` performs this). When implementing this, there is a tradeoff between time and space, which allows for 2 implementations of this design: a combinatorial implementation, where each component is only used once and the components are layered out in space, with the components for each iteration receiving the outputs of the last iteration as inputs; and a sequential implementation, where a register is used for each datum (the accumulator and the shifter) and there is a single copy of each hardware component, but several clock cycles are needed to complete operation. A combinatorial design will tend to have lower latency and lower information leakage, since the components used are relatively simple and easy to parallelize and the main leakage component comes from the registers storing the data, but it costs more area, power and price, and tends to be harder to analyze since its operation is too fast and its leakage is too low for a scope to track with a relatively small number of traces.

In our main RAMBAM module and the simplest S-Box implementation, we included a combinatorial implementation of the multiplier module. However, for better analysis and comparison we also created a sequential implementation.

Note that however the multiplier here is implemented, there is no randomization refresh between the atomic operations – shifting $x^i p_1(x)$ and adding to the accumulator are done without adding a random codeword after them. This suggests a vulnerability in

the architecture, see section 4.4. However, adding refresh randomization to this design requires an addition of $2(m + d)$ d-bit random inputs and an additional module for encoding r *per multiplication*, leading to a significant increase in cost and randomization usage.

The improvements in CLM suggest another multiplier implementation, in which the reduction is performed after the multiplication via systematic encoding, and the significance of Q in the reduction is eliminated. For further elaboration, see section 5.1.

4.3.5 Underlying System Architecture

All cryptographic modules were implemented and tested on a SASEBO-GIII [19] board (licensed as Sakura X), with framework code provided by Japan's AIST and modified to fit the size of the inputs and outputs needed by the modules. The board contains 2 FPGAs – a master chip whose role is to communicate directly to the computer, and a slave chip which is the cryptographic target. The master chip sends and receives data through an FTDI FT2232H chip located on-board; this chip connects to the computer using a USB interface. The master and slave chips communicate via a 16-bit local bus. The chips provide the modules with global clock and reset signals, which are used by all sequential modules.

Synthesis and implementation were done using the Xilinx Vivado toolset, and loading the bitstreams onto the FPGAs were done using Xilinx's IMPACT tool. Hardware simulations were done both using Vivado and using Cadence's XCelium & Simvision simulators.

Inputs to the cryptographic algorithms are generated by a computer program, then sent through this interface to the target chip. These inputs include the random bits used by the masking scheme, which are generated by the program using a cryptographically-strong RNG. When all inputs to an algorithm are prepared, a trigger pin located on-board is switched by the target chip, and the computation statrts. When it is finished, the output of the computation is sent back to the computer. Inputs for consecutive encryptions are sent one by one: the inputs to the an encryption are only sent from the computer to the hardware once the outputs of the previous encryption are received by the computer, and the trace collection for it finishes. For any measurement on an S-Box, the input is generated as an 8-bit string which is then randomized in software using a d-bit random input multiplied by the instance's P.

The power consupmtion of the target chip is measured using a PicoScope® 5000-series oscilloscope which is triggered by the target chip's trigger pin, with the sample rate,

sample count and resolution defined in software to best match the measurements.

4.4 Suggested Weaknesses & Attacks

As noted above, there is no randomization refresh after every atomic operation in the multiplier. This causes bits from the multiplicand p_1 to intermix within the shifter, and possibly – if a particularly bad choice of Q is made, like $Q(x) = x^d$ – partly or entirely eliminate the added randomness within p_1 after enough shifts, revealing the hidden internal algorithmic datum. Additionally, the multiple consecutive operations that are controlled by the bits of p_2 may reveal p_2 entirely or partially – from which the internal datum can be easily recovered by reduction modulo P. These weaknesses suggest that with enough traces, and a sequential implementation of the multiplier for "easier access" to internal values, even a simple fixed-vs-fixed test on the multiplier can succeed.

Additionally, it can be noted that there is no refresh present after the MixCols step. Though this may be hard to exploit or notice in a t-test, it means that the operands to the MixCols step are re-used in later steps of the cipher without re-randomizing them – and leaks information about them.

Though this is likely an oversight, the key input to the cipher is not randomized using random multiples of P. Therefore, in an implementation that does not precompute all the round keys, the key may leak from the S-Boxes computing the key-expansion algorithm. Since this can easily be solved by randomizing the key as well as the plaintext, we ignore attacks stemming from this vulnerability.

A naïve comparison of d with the definition of masking order in boolean masking suggests that using a RAMBAM module parameterized by d with full randomness and all other vulnerabilities resolved should protect against t-tests targeting the $\frac{d}{8}$-th moment or below. A useful experiment would be to use d leading to a fractional masking order (such as $d = 4$), and observe the leakage in the various moments and whether there is an improvement in the lower moments compared to a d that leads to a whole masking order. Another experiment that may be run is not "using the full amount" of randomness in an implementation: implementing a RAMBAM instance with a given d, but feeding it random inputs with less than d bits of randomness, e.g. polynomials of degree $< d' < d$. It can be checked whether the reduced randomness can still bring about adequate protection, or whether it instead provides reduced protection in comparison to a system designed for less randomness. An edge case of this is comparison with an off-the-shelf AES module with no protection, when compared to a RAMBAM module with all random

inputs fixed.

Since both the polynomials $f_1(x) = 0$ and $f_2(x) = 1$ are fixed points of both the input and output transformations and the exponentiation in the S-Box, they lend themselves well as candidates for the fixed value in a fixed-vs-random test. An additional good candidate is choosing a random input of Hamming weight $m/2 = 4$ (the mean of the input Hamming weight distribution).

Results of evalutaions on the tests detailed above are detailed below.

4.5 Evaluation Results

Figure 4.3 shows the implementation cost in FPGA LUTs of a single RAMBAM S-Box, when placed within the hardware framework described in section 4.3.5, as a function of the security parameter d. As expected, a roughly quadratic increase in d can be seen. It can be also seen that there is a strong dependence of the cost on the choice of P and Q.

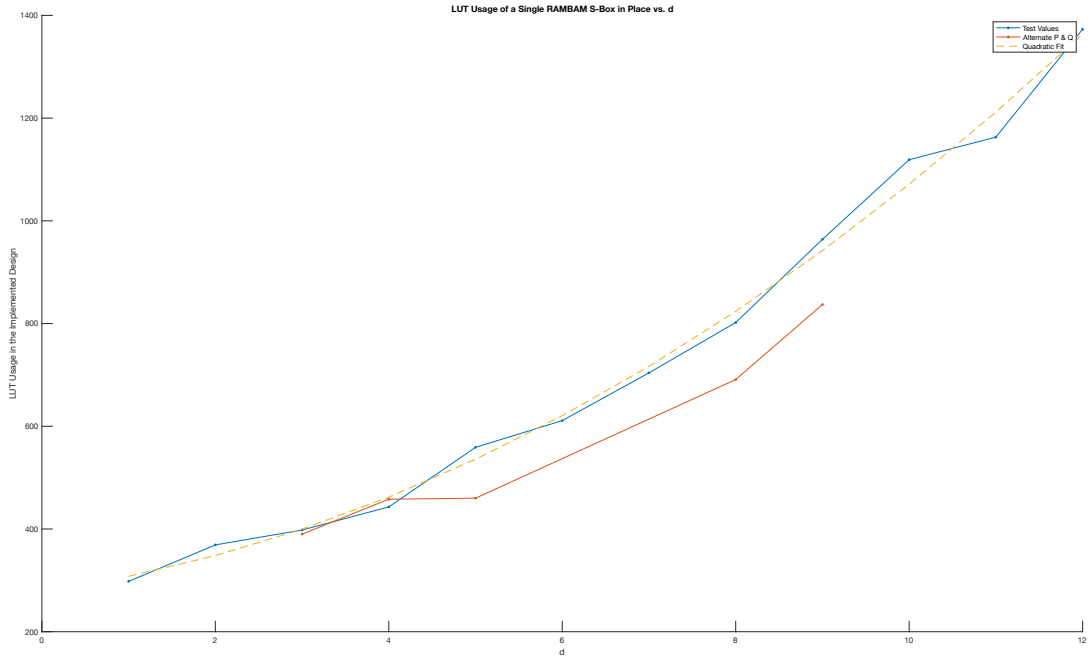


Figure 4.3: Implementation Cost of a Single RAMBAM S-Box vs. d

All measurements performed for the sake of the figures below were with a sample frequency of $500MS/s$ and a bit resolution of 14 bits/sample. The scope's measurement

input was connected to a gain-10 amplifier connected serially to the chip's power consumption output, and was set to have a range of $\pm 0.1V$.

Figure 4.4shows a mean trace of 10000 traces captured from a single S-Box implementation: the 7 operating cycles can be clearly seen, with the entire operation taking about 300 sample points. Figure 4.5 shows a mean trace of 20000 traces captured from an implementation of RAMBAM: the 10 AES rounds and their components can be clearly idengtified from the trace, with the entire operation taking about 5000 samples points.

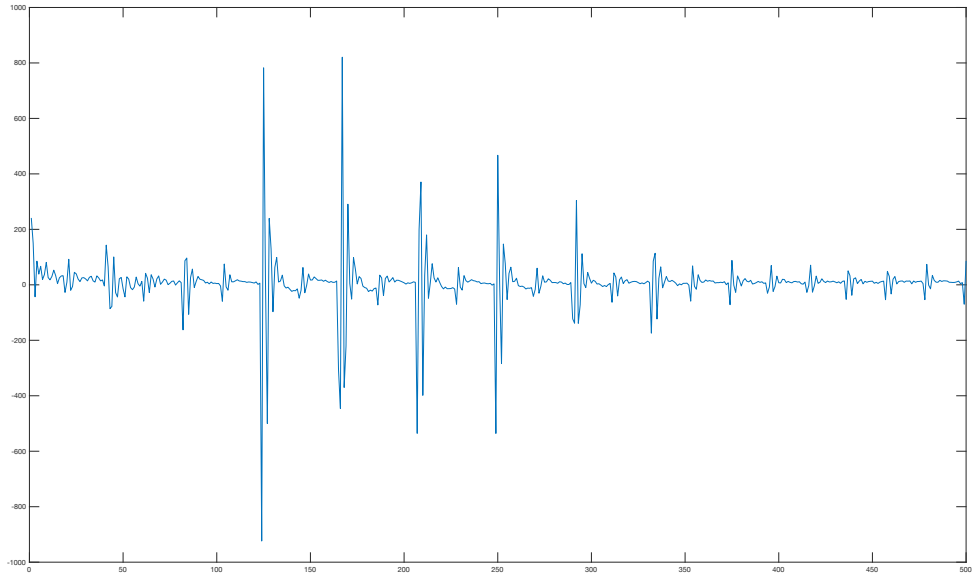


Figure 4.4: Mean S-Box Trace

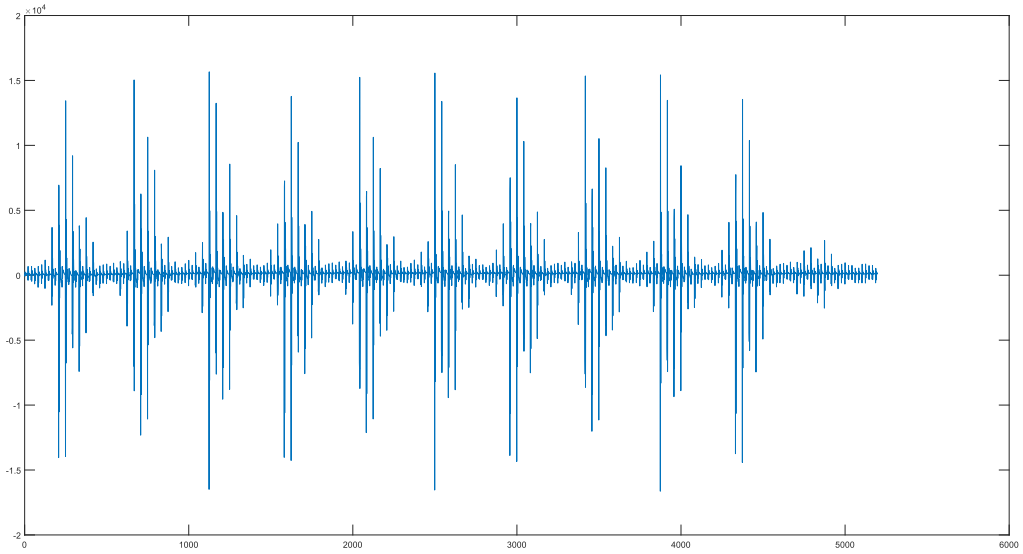
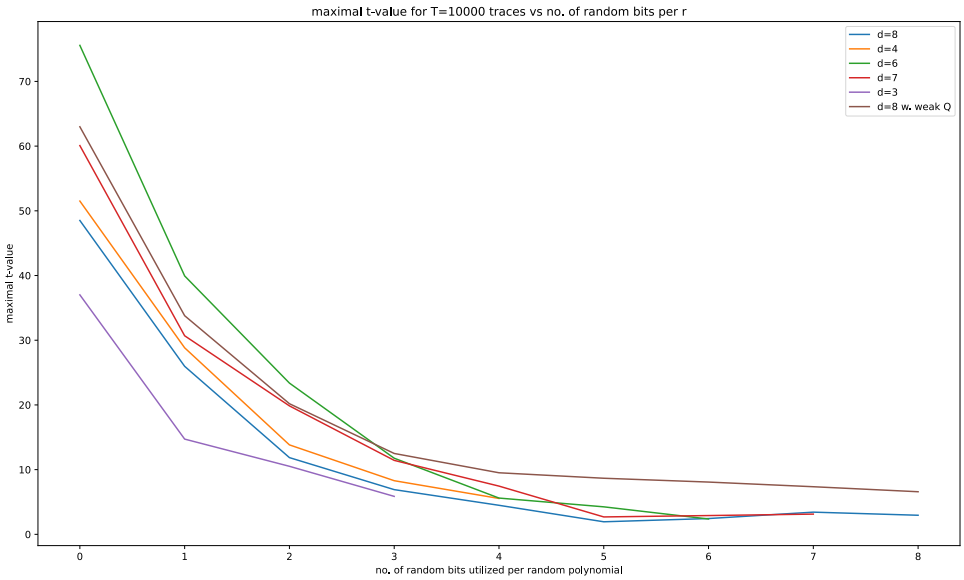
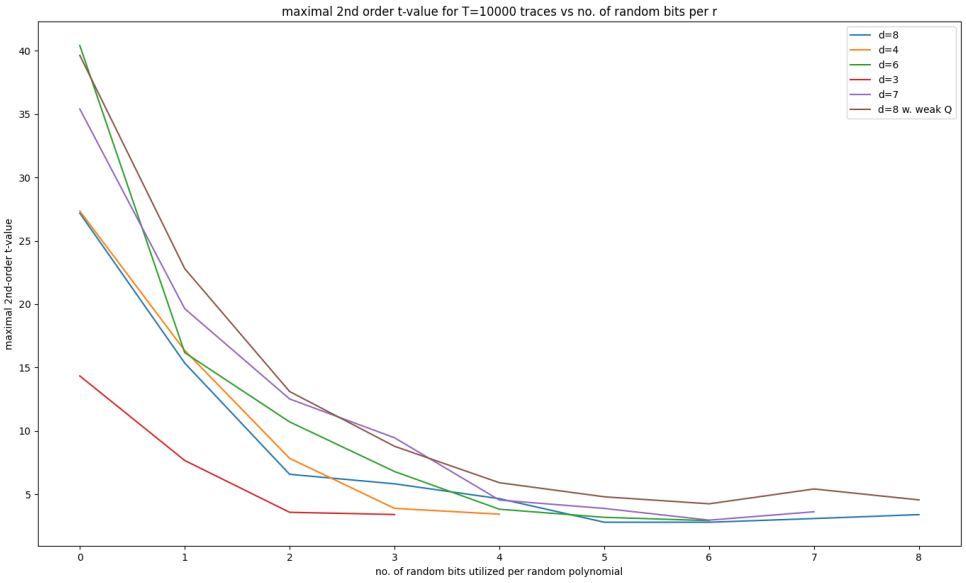


Figure 4.5: Mean RAMBAM Trace

Figure 4.6a shows the maximal t-value obtained in an FvR test with $T = 10000$ traces captured from an implementation of a single S-Box; these are plotted as a function of the number of random bits utilized per random input r_i. Multiple graphs are shown, each corresponding to a different implementation: the default choice of parameters was taken directly from the RAMBAM paper. Figure 4.6b shows the same result under a 2nd order t-test. From these graphs, several conclusions may be drawn: the first is that the security of the scheme is highly dependent on d, P and Q. The second is that even values of d that do not give a masking order that fully protects against a test on a particular statistical moment still does partially protect against it. A third is that even usage of limited randomness – with each r_i having j random bits – helps partially, in the best cases roughly equivalently to usage of an implementation with $d = j$ and all random bits.



(a) 1st order t-test



(b) 2nd order t-test

Figure 4.6: Maximal t-value vs. randomness utilized for different d

Figure 4.7 shows the maximal t-value obtained in an FvR test for several different fixed plaintexts as a function of the number of traces, up to $750k$. The traces were taken from an implementation of a single S-Box with $d = 8$, utilizing the maximal amount of

randomness. From the figure it can be seen that as hypothesized in section 4.4, the t-test results depend on the plaintext input of the S-Box: there are fixed plaintexts that are more vulnerable than other fixed plaintexts. Howver, contrary to the hypothesis, the plaintext found to be more vulnerable was not 0 or 1 but rather a weight-4 plaintext. It can also be seen that the S-Box is, indeed, vulnerable: at very high trace counts the t-test does increase, and it surpasses the threshold for being statistically significant.

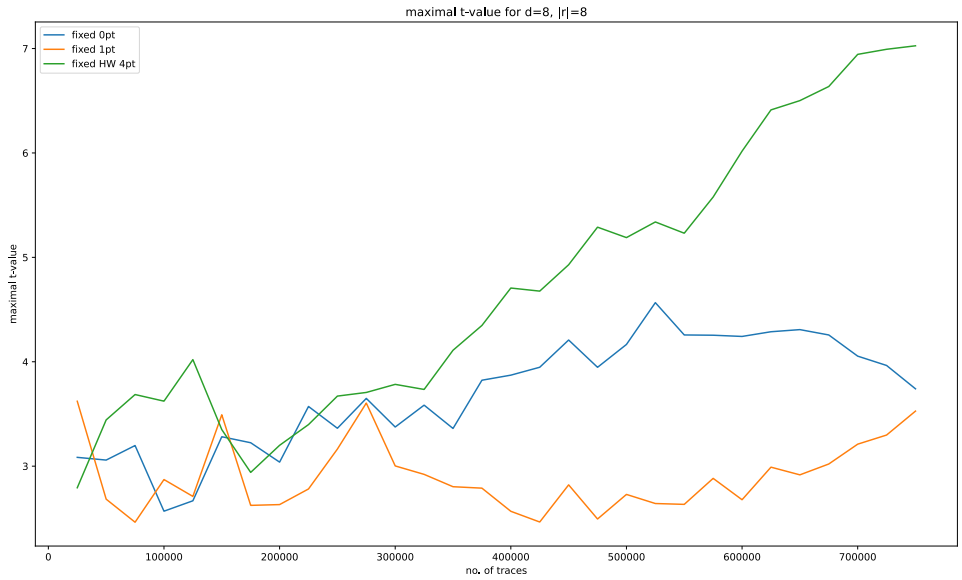


Figure 4.7: Maximal t-value vs. Number of Traces for Several Plaintexts

Figure 4.8 shows the maximal t-value obtained in an FvR test for the entire RAMBAM system, as a function of the number of traces, up to $20k$. From the figure it can be seen that at $d = 4$ (partial masking order), the result of the t-test is high and increases rapidly, even with full randomness. With $d = 8$ and partial randomness (4 bits per r_i), the result is rougly the same as with $d = 4$ and full randomness, very similarly to the result in figure 4.6a. With $d = 8$ and full randomness, however, the t-test result is low and stays low for the entire trace count.

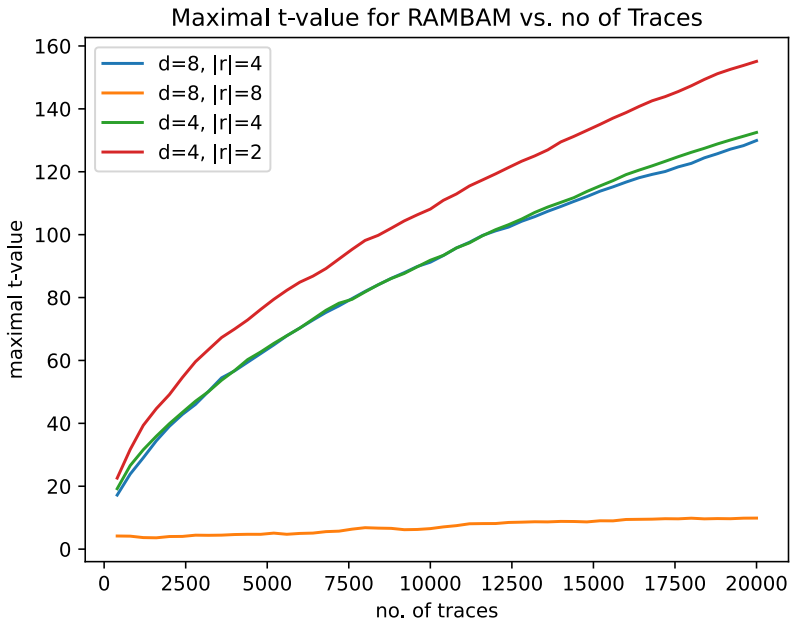


Figure 4.8: Maximal t-value vs. Number of Traces for the Entire RAMBAM System

4.5.1 Systematic vs. Convolution Encoders

As a sample comparison between 2 implementation alternatives we compared between implementations of the S-Box utilizing a convolutional encoder for randomization and ones utilizing a systematic encoder for randomization. Figure 4.9 shows the maximal t-value between traces extracted from both S-Box implementations, with both acting on a fixed plaintext taken from an FvR test. Different plots correspond to both different implementations (i.e. a change in d), and a different amount of randomness used. The test shows a clear distinction between the traces taken from the 2 implementations, outlining the difference in architecture.

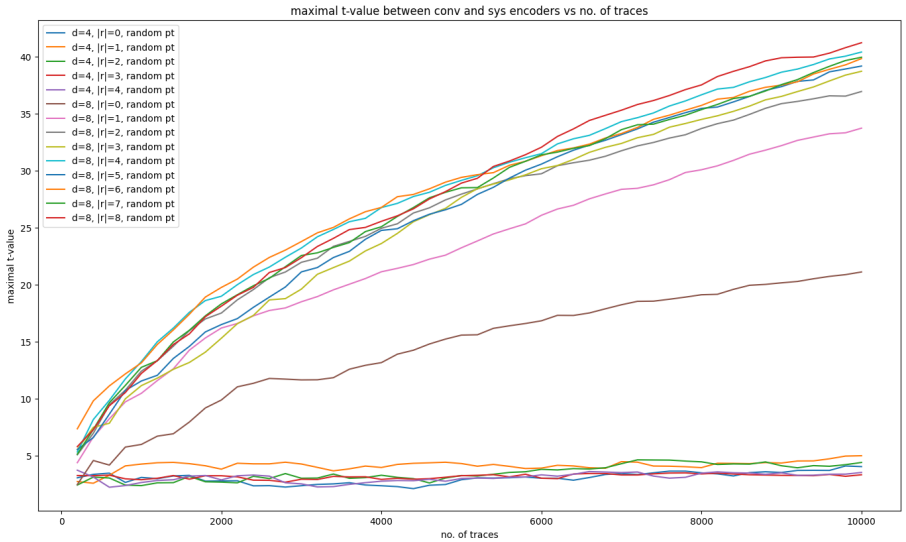
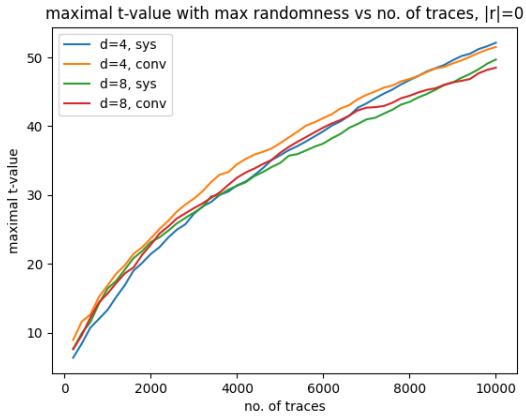
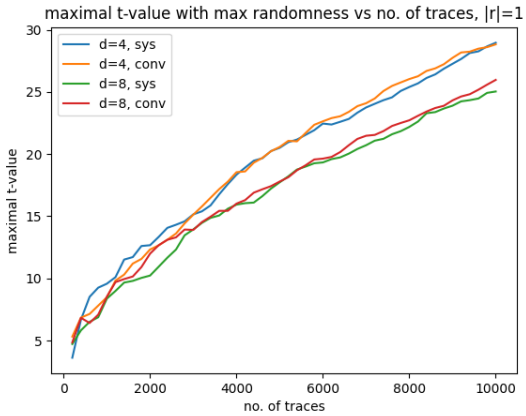


Figure 4.9: Maximal t-value between Conv. & Sys. Encoder vs. Number of Traces for

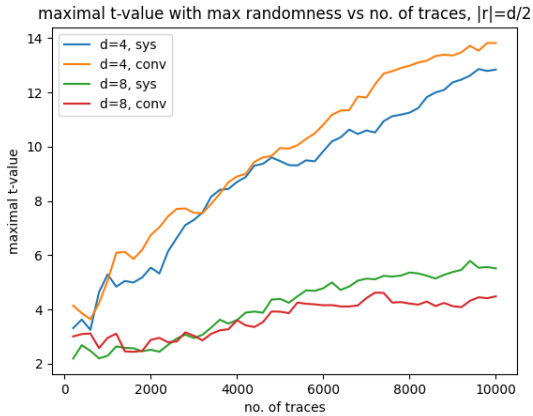
Figures 4.10a–4.10d show the maximal t-value from an FvR test on each of the 2 types of implementations as a function of the number of traces. Different figures correspond to different amounts of randomness (denoted by $|r|$) utilized. While a slight advantage appears towards one of the implementations in 4.10b and 4.10c, this advantage is not consistent among different fixed plaintexts and different amounts of utilized randomness – suggesting no implementation is clearly favorable over the other.



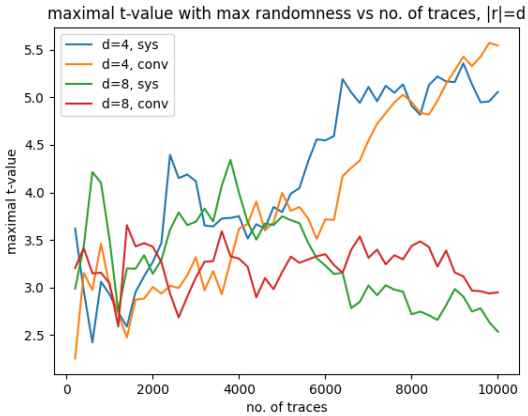
(a) t-value with $|r| = 0$



(b) t-value with $|r| = 1$



(c) t-value with $|r| = d/2$



(d) t-value with $|r| = d$

Figure 4.10: Maximal t-value vs. Number of Traces On Systematic & Convolutional Encoders with Varying $|r|$

5 Consolidated Linear Masking

5.1 The Premise & Comparison to RAMBAM

CLM is a masking scheme, introduced in [13], that is a suggested improvement to RAMBAM. Its main idea is that additional randomness can be added with a minimal increase in hardware cost by treating P as an additional input to the cipher, and randomizing it as well as the polynomials multipliying it. Randomizing P means randomizing L, which means each field element will possibly have more representations than in the original RAMBAM. From Gauss' irreducible polynomial counting lemma, there are 30 irreducible polynomials of degree 8 over GF(2), each having 8 roots which can be chosen to generate L – for a total .

To formalize the advantage given by randomizing L, the paper formulates a definition for masking order based on entropy: given an RV V representing the original input, distributed uniformly between all possible inputs, and an RV U representing its masked representation, the masking order is defined as

$$d = \frac{H(U \mid V)}{H(V)}$$

that is, how much randomness is "added" divided by how much randomness exists in the original datum. Using this definition, the masking order is increased by $\frac{H(VL|V)}{H(V)}$ – which is *less than* $\log_2(30)$ (or $\log_2(240)$ when randomizing a root) since some values can have only a small amount of images under the different Ls: for example, 0 must be mapped to 0 under any L.

The scheme also suggests eliminating the reliance on Q entirely, by showing that it can be eliminated via a correct use of refreshes: each part which appears reliant on Q in the RAMBAM cipher can be shown to be reliant only on P and d, if a certain implementation of the cipher is made. With this implementation, randomizing P does not compromise the part of the security that is reliant on Q. A list of the parts for which this is not obvious is detailed below.

1. In the affine transformation, the component matrices of T are given by $T_{11} = L^{-1}WL$ and the condition $BT_{1,1} + T_{2,1} + BT_{22} = 0$, and the vector t is given by the condition $wL = tH^T$. Though the matrices and the vector may be chosen to rely on Q, they may perfectly well, without weakening security, be chosen to rely on P and only.

2. In all 3 instances where a computation is performed modulo PQ – in the raising

to the power and multiplication steps of the S-Box, and in the MixCols step, when a multiplication by $L(2)$ is performed, a reduction modulo PQ can be shown to be redundant if a refresh of the randomness is made. Let $w = \begin{pmatrix} w_0 & \dots & w_{n-1} \end{pmatrix}$ be an $n > m + d$-bit word that we wish to reduce modulo PQ. This amounts to adding to $w(x)$ a polynomial $A(x) = a(x)P(x)Q(x)$ with $\deg a = n - m - d$, subject to the condition $(A_{m+d} \dots A_{n-1}) = (w_{m+d} \dots w_{n-1})$. After adding a refresh in the form of $R(x) = r(x)P(x)$, the resultant polynomial is

$$w(x) + a(x)P(x)Q(x) + r(x)P(x) = w(x) + B(x)P(x).$$

$B(x)P(x)$ is a codeword of the shortened cyclic code of length n and dimension $n - m$ which is constrained by the $n - m - d$ conditions above, leaving d degrees of freedom: these can be determined by Q, but they can just as well be determined by r, since r has d degrees of freedom. Using r thus completely eliminates the modular reduction's dependence on Q.

Point 2 suggests alternative implementations for the multiplier, power and MixColumns modules, in which modular reduction and refresh are done in tandem and only at the end of the series of operations. This is done by using a systematic encoder as seen in section 4.3.1 for the code generated by P and with length of either $2(m+d)-1$ (for the multiplier and power modules) or $2m+d-1$ (for the MixCols module) to generate $B(x)P(x)$, then adding the result to the result of the operation pre-modular reduction. To avoid dealing with inordinately high powers of x pre-reduction, the power computation must be done via repeated squaring.

5.2 Implemetation Details

When implementing CLM, an important point is the way in which the additional randomness specifying P and Q is fed to the system. Since the considerations from the original RAMBAM on the choice of P are largely removed, and Q becomes redundant entirely, any irreducible P may be chosen for an encryption cycle. As noted above, there are either 30 or 240 options when specifying P. This means that specifying P directly using 9 input bits is redundant, since only a certain subset of polynomials of degree 8 can be used. Using the limited amount of permitted polynomials, P may either be specified using 5 bits to enumerate the polynomial, or 8 bits for to enumerate the root. To start, we chose an implementation using 5 bits of randomness to specify P .

Our implementation of CLM was largely based on our implementation of RAMBAM.

The fundamental architecture remains the same, and the most significant changes from the original was the addition of an input for specifying P, and and an addition of a clock cycle for parameter extraction at the beginning of each encryption operation. In this clock cycle, the input specifying P is fed into a parameter extraction submodule which calculates each of the parameters P, L, L^{-1}, M_{conv}, B (and its extended versions for $2(m+d)-1$ and $2m+d-1$ rows), $T_{1,1}$, $T_{2,1}$, and t. These are then stored in a `params` object, which is fed as an input in later clock cycles to all the internal submodules which utilize these parameters.

As in section 5.1, the implementation of CLM presents a choice on how exactly to extract the parameters specified above from the input specifying P. For any of the above parameters, one approach is to use a specific hardware circuit specifically tailored to generate it – such as a circuit using shifted copies of P to generate M_{conv}; another is to use a memory module, where parameter choices for each value of P and d are generated offline via a software module then stored at compilation time in memory: the correct value is then extracted at runtime with the input specifying P being the input to the memory module. Ultimately, since these parameters are not fixed at runtime and both methods leak information about the random inputs, this is mainly a question of hardware efficiency vs. memory use. Our choice in this matter was to use a memory module for P, L, L^{-1}, T and t, and a hardware generator for M_{conv}, B and its extended variants. Since T_{21} and t both depend on d, a different memory file is needed for each d.

The MixCols, power and multiplier modules were changed to the implementations suggested in section 5.1, with the power module implemented via a series of squaring operations laid out one after another combinatorially, and the randomness used for refresh re-used across all squaring operations needed for a single power raise. The randomness used in the MixCols modular reduction step is re-used from the randomness applied to the inputs in the first stage of the cipher.

5.3 Results

Figure 5.1 shows the implementation cost in FPGA LUTs of RAMBAM and CLM implementations when placed within the hardware framework described in section 4.3.5, as a function of the security parameter d.. Since, as noted above, the implementation cost of RAMBAM highly depends on the choice of P and Q, for comparison we chose the optimal parameters suggested in the original RAMBAM paper. We have yet to functionally verify the CLM implememtation, and so these results are unverified and untested. This figure suggests the implementation of CLM is significantly cheaper than the implemen-

tation of RAMBAM, even though they both use the same architecture, and increases roughly linearly with d: this result is surprising and needs to be further tested.

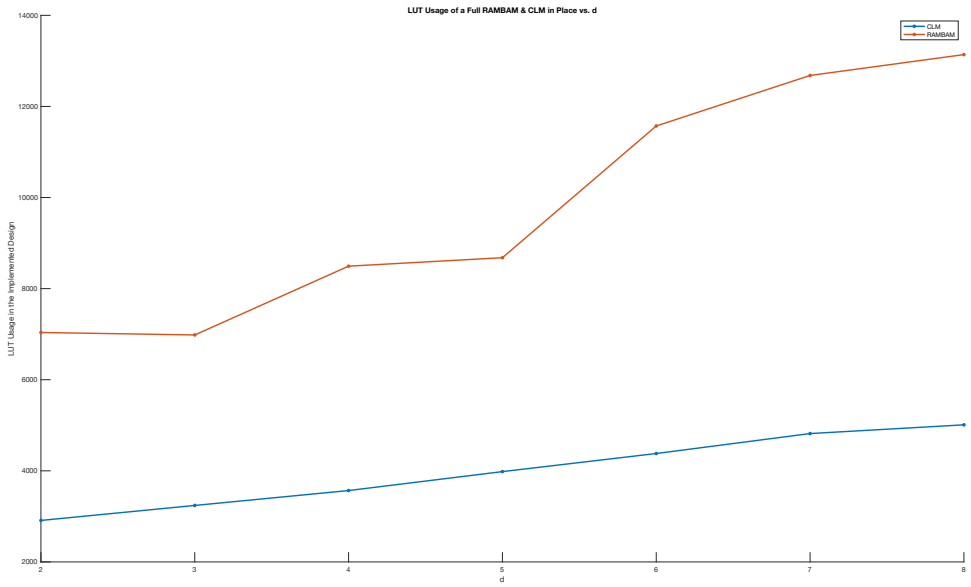


Figure 5.1: Implementation Cost of RAMBAM and CLM vs. d

6 Summary & Future Work

In this work, 2 masking schemes were implemented and tested. The 2 masking schemes are specifically tailored to the AES-128 encryption scheme, and aim to protect it from power analysis attacks. They both use a form of linear masking wherein the field elements used in encryption are given redundant representations with an additional d bits, and random multiples of a polynomial are used to mask the hidden value. The first scheme, RAMBAM, operates with fixed masking polynomials and performs operations in a fixed ring. The second scheme, CLM, generalizes on RAMBAM by randoming the masking polynomial as well as the un-masked representation of each field element, and eliminates the need to perform operations in a ring via a clever use of re-randomization. The 2 schemes have an advantage of granularity over classical masking schemes, since the order of protection they offer is not limited to integer multiples.

Implementation of the 2 schemes shows that the implementation cost of RAMBAM increases roughly quadratically with the masking order, and is dependent on the ring chosen. The protection offered by RAMBAM increases with the amount of randomness used and with d, and is also dependent on the choice of the ring. We showed that as expected, the scheme as proposed in the original paper has weaknesses stemming from poor use of re-randomization and that result in it failing in high trace counts; and that its security is inherently dependent on the specific plaintext chosen to encrypt. The cost of CLM also increases with d, and appears at first glance to be less than that of RAMBAM.

There remains much further research to be done on the 2 masking schemes. Alternative implementations to RAMBAM that attempt to fix the original's shortcomings may be designed, tested and contrasted with the suggested implementation. Further analysis may be performed to ascertain the precise considerations required when choosing the correct polynomial and ring, and to find the precise relation between d and the protection the scheme offers against attacks on the nth order moment.

For CLM, the precise cost of the scheme relative to a parallel implementation of RAMBAM needs to be further checked. An implementation utilizing the full amount of randomness inherent in the unmasked representation may be designed and contrasted with the current implementation. A comprehensive side-channel analysis on the implementation of CLM needs to be done, and its protection needs to be compared to the protection RAMBAM offers. It may be checked whether the inherent weaknesses in RAMBAM persist in CLM, and whether its claims on the redundancy of the ring are true.

References

[1] P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi, "Introduction to differential power analysis," *Journal of Cryptographic Engineering*, vol. 1, no. 1, pp. 5–27, Apr 2011. [Online]. Available: https://link.springer.com/content/pdf/10.1007/s13389-011-0006-y.pdf

[2] E. Brier, C. Clavier, and F. Olivier, "Correlation power analysis with a leakage model," in *Cryptographic Hardware and Embedded Systems - CHES 2004*, M. Joye and J.-J. Quisquater, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 16–29.

[3] G. Ratanpal, R. Williams, and T. Blalock, "An on-chip signal suppression countermeasure to power analysis attacks," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 3, pp. 179–189, 2004.

[4] A. Gornik, A. Moradi, J. Oehm, and C. Paar, "A hardware-based countermeasure to reduce side-channel leakage: Design, implementation, and evaluation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 8, pp. 1308–1319, 2015.

[5] T. Güneysu and A. Moradi, "Generic side-channel countermeasures for reconfigurable devices," in *Cryptographic Hardware and Embedded Systems – CHES 2011*, B. Preneel and T. Takagi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 33–48.

[6] Z. Chen and Y. Zhou, *Dual-Rail Random Switching Logic: A Countermeasure to Reduce Side Channel Leakage*. Springer Berlin Heidelberg, 2006, p. 242â 254. [Online]. Available: http://dx.doi.org/10.1007/11894063_20

[7] T. Popp and S. Mangard, "Masked dual-rail pre-charge logic: Dpa-resistance without routing constraints," in *Cryptographic Hardware and Embedded Systems – CHES 2005*, J. R. Rao and B. Sunar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 172–186.

[8] A. Razafindraibe, M. Robert, and P. Maurine, "Improvement of dual rail logic as a countermeasure against dpa," in *2007 IFIP International Conference on Very Large Scale Integration*, 2007, pp. 270–275.

[9] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi, "Towards sound approaches to counteract power-analysis attacks," in *Advances in Cryptology — CRYPTO' 99*, M. Wiener, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 398–412.

[10] O. Keren and I. Polian, "Ipm-red: combining higher-order masking with robust error detection," *Journal of Cryptographic Engineering*, vol. 11, no. 2, pp. 147–160, 2021. [Online]. Available: https://doi.org/10.1007/s13389-020-00229-4

[11] E. Prouff and M. Rivain, "Masking against side-channel attacks: A formal security proof," in *Advances in Cryptology – EUROCRYPT 2013*, T. Johansson and P. Q. Nguyen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 142–159.

[12] Y. Belenky, V. Bugaenko, L. Azriel, H. Chernyshchyk, I. Dushar, O. Karavaev, O. Maksimenko, Y. Ruda, V. Teper, and Y. Kreimer, "Redundancy aes masking basis for attack mitigation (rambam)," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, Feb. 2022. [Online]. Available: http://dx.doi.org/10.46586/tches.v2022.i2.69-91

[13] I. Levi and O. Keren, "Consolidated linear masking (CLM): Generalized randomized isomorphic representations, powerful degrees of freedom and low(er)-cost," Cryptology ePrint Archive, Paper 2024/967, 2024. [Online]. Available: https://eprint.iacr.org/2024/967

[14] G. Goodwill, B. Jun, J. Jaffe, and P. Rohatgi, "A testing methodology for side channel resistance validation," 2011. [Online]. Available: https://api.semanticscholar.org/CorpusID:16852899

[15] T. Schneider and A. Moradi, *Leakage Assessment Methodology: A Clear Roadmap for Side-Channel Evaluations*. Springer Berlin Heidelberg, 2015, pp. 495–513. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-48324-4_25

[16] Y. Ishai, A. Sahai, and D. Wagner, "Private circuits: Securing hardware against probing attacks," in *Advances in Cryptology - CRYPTO 2003*, D. Boneh, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 463–481.

[17] J. Daemen and V. Rijmen, *The Design of Rijndael: The Advanced Encryption Standard (AES)*. Springer Berlin Heidelberg, 2020. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-60769-5

[18] Fortify-IQ, inc. (2023) FIQ-OpenAES-128e - A Sample Implementation of RAMBAM. [Online]. Available: https://github.com/fortify-iq/fiq-openaes-128e/

[19] Y. Hori, T. Katashita, A. Sasaki, and A. Satoh, "Sasebo-giii: A hardware security evaluation board equipped with a 28-nm fpga," in *The 1st IEEE Global Conference on Consumer Electronics 2012*, 2012.

A State Machines for the RAMBAM module

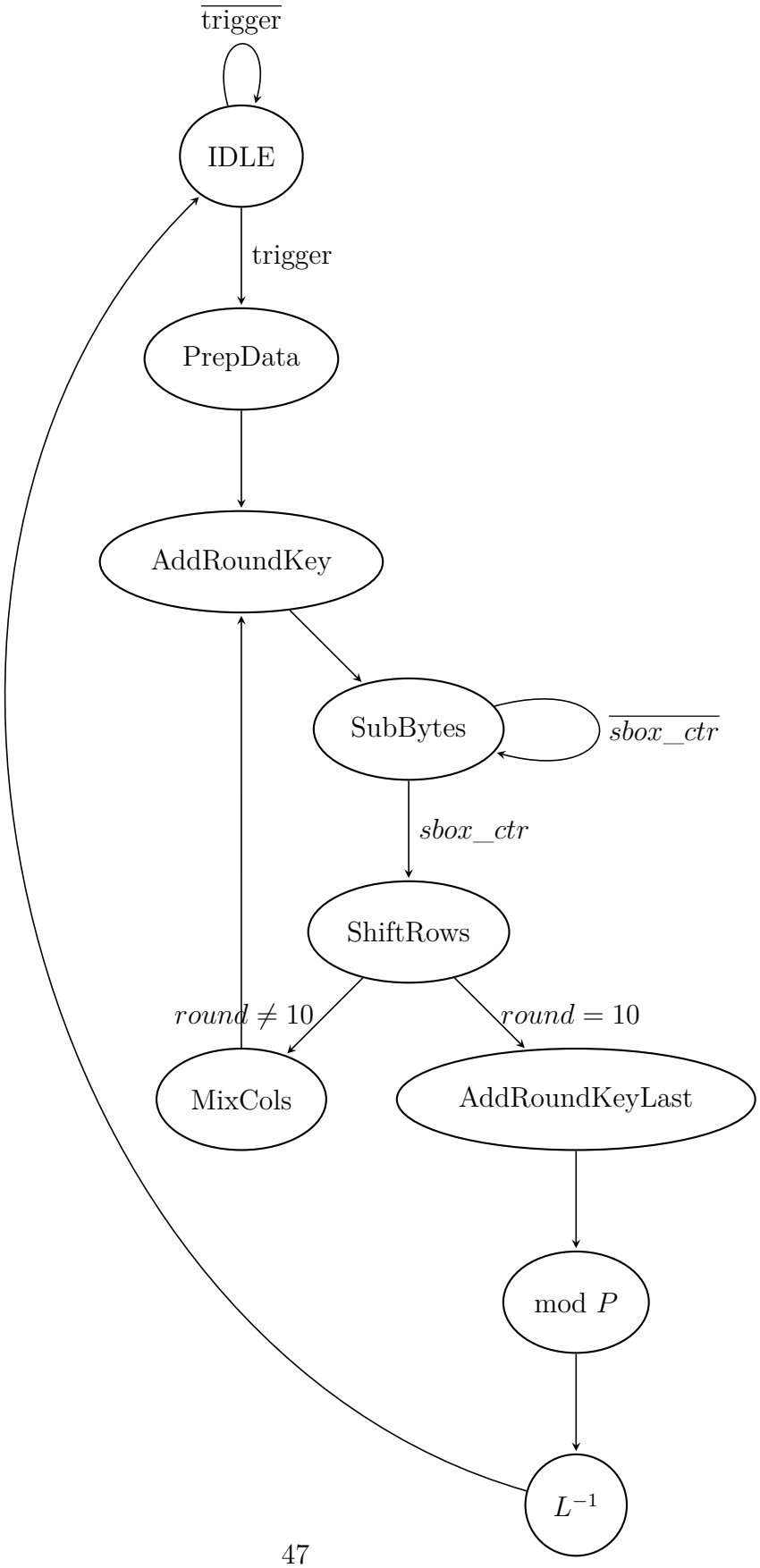


Figure A.1: State Machine for the RAMBAM Module

Here the input perparation state multiplies the plaintext and key by L, and adds randomness to the plaintext. "Trigger" is an external trigger passed from the top level communication module, and the S-Box Counter counts clock cycles in the S-Box operation.

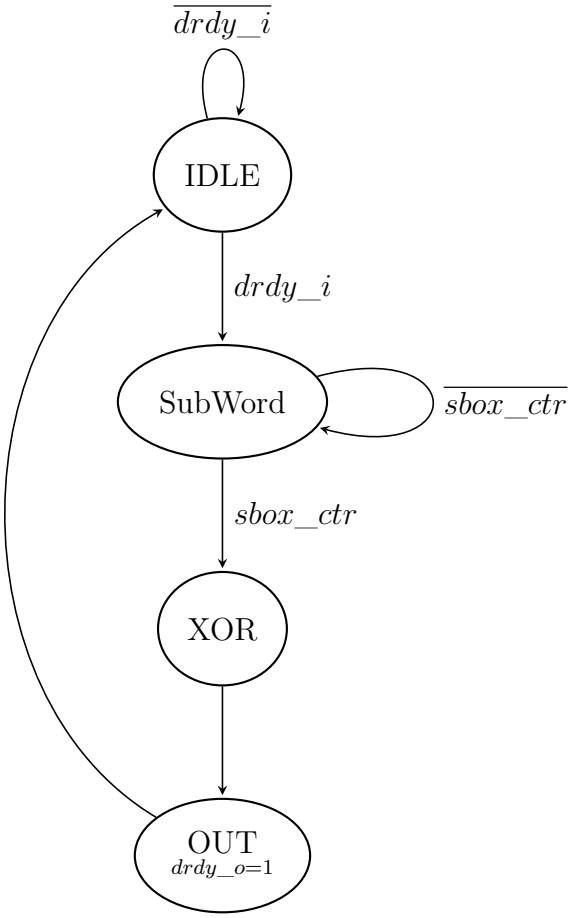


Figure A.2: State Machine for the RAMBAM Key Expansion Submodule

Here the XOR state is where the output is prepared. drdy_i and drdy_o are control signals that originate in the top-level RAMBAM module. The S-Box counter here is separate from the S-Box counter held by the top-level RAMBAM module.

תקציר

מיסוך הוא אמצעי המנע הנפוץ ביותר כיום נגד מתקפות הספק. סכמות מיסוך הקיימות כיום נוטות להיות יקרות ולגרום לתקורה גבוהה בשטח וברנדומיות. במטרה לתקן זאת, 2 סכמות מיסוך חדשות הוצעו. סכמות אלו מותאמות לאלגוריתם ההצפנה הנפוץ AES, ולהן מחיר נמוך ושליטה הדרגתית יותר בסדר המיסוך מלסכמות מיסוך קלאסיות. הראשונה, רמב"ם, מעבירה את הקלטים להצפנה לייצוג שקול באמצעות L, ואז ממסכת אותם באמצעות כפולות רנדומיות ממעלה קטנה מ-d של פולינום P היוצר את הייצוג השקול. פעולות ברמב"ם מבוצעות בחוג R_{PQ} שמכיל את שדה הייצוגים השקולים של כלל האיברים. הסכמה השנייה, סכמת מיסוך לינארית מוכללת, מוסיפה לרמב"ם רנדומיות נוספת במחיר נמוך על ידי כך שהיא מרנדמת גם את P ואת L. פעולות בה מבוצעות באמצעות סוג של אריתמטיקת ריענונים שמחליפה הורדה מודולו PQ, ובכך מייתרת את התלות ב-Q.

שתי סכמאות המיסוך ממומשות, ועלותן מוצגת כתלויה ב-d, ובמקרה של רמב"ם, גם ב-P וב-Q. רמב"ם, במימושו המוצע, מוצג כפגיע למתקפות ערוצי צד בכמות טרייסים גבוהה, כאשר ערכי הצפנה מסויימים פגיעים יותר מערכים אחרים. תוצאותינו מראות כי למרות זאת, הסכמה מציעה הגנה חלקית גם כאשר $d/8$ נמוך מסדר המתקפה וגם כאשר נעשה שימוש בכמות מוגבלת של רנדומיות.

אוניברסיטת בר-אילן
Bar-Ilan University

הפקולטה להנדסה

המעבדה להגנת סייבר

מימוש חומרה וניתוח ערוץ-צד של סכמת מיסוך מבוססת יתירות וסכמת מיסוך לינארית מוכללת עבור AES

יאיר אורן

דן מיכאלי

פרויקט שנה ד' לקראת תואר ראשון בהנדסה

מנחים אקדמיים: פרופ' איתמר לוי ופרופ׳ אסנת קרן

אוקטובר 2024