A New Framework for Constraint-Based Probabilistic Template Side Channel Attacks

Yossef Oren[1], Ofir Weisse[2], and Avishai Wool[3]

[1] Network Security Lab, Columbia University, USA
[2] School of Computer Science, Tel-Aviv University, Israel
[3] School of Electrical Engineering, Tel-Aviv University, Israel
yos@cs.columbia.edu, ofirweisse@gmail.com, yash@eng.tau.ac.il

Abstract. The use of constraint solvers, such as SAT- or Pseudo-Boolean-solvers, allows the extraction of the secret key from one or two side-channel traces. However, to use such a solver the cipher must be represented at bit-level. For byte-oriented ciphers this produces very large and unwieldy instances, leading to unpredictable, and often very long, run times. In this paper we describe a specialized byte-oriented constraint solver for side channel cryptanalysis. The user only needs to supply code snippets for the native operations of the cipher, arranged in a flow graph that models the dependence between the side channel leaks. Our framework uses a soft decision mechanism which overcomes realistic measurement noise and decoder classification errors, through a novel method for reconciling multiple probability distributions. On the DPA v4 contest dataset our framework is able to extract the correct key from one or two power traces in under 9 seconds with a success rate of over 79%.

Keywords: Constraint solvers, power analysis, template attacks.

1 Introduction

In a constraint-based side-channel attack, the attacker is provided with a device under test (DUT) which performs a cryptographic operation (e.g., encryption). While performing this operation the device emits a data dependent side-channel leakage such as power consumption trace. As a result of the data dependence, a certain number of leaks are modulated into the trace together with some noise. In order to recover the secret key from a power trace the attacker performs the following steps:

Profiling: The DUT is analyzed in order to identify the position of the leaking operations in the traces, for instance by using classical side-channel attacks like CPA [4]. Then a decoding process is devised, that maps between a single power trace and a vector of leaks. A common output of the decoder is the Hamming weight of the processed data as in [22], but many other decoders are possible. An effective profiling method is a template attack, which was introduced in [5]. Profiling is an offline activity.

Decoding: After the profiling phase, the attacker is provided with a small number of power traces (typically, a single trace). The decoding process is applied

L. Batina and M. Robshaw (Eds.): CHES 2014, LNCS 8731, pp. 17–34, 2014.

to the power trace, and a vector of leaks is recovered. This vector of leaks may contain some errors, e.g., due to the effect of noise.

Solving: The leak vector, together with a description of the algorithm implemented in the DUT, and additional auxiliary information, is converted to a representation that is suitable to a constraint solver: e.g., a SAT-solver [21,22,28] or a Pseudo-Boolean solver [17,18]. The solver solves the problem instance, outputting the best candidates satisfying the constraints. However, previously used solvers require a bit-level representation which creates several challenges. In this paper we suggest a new solver which uses a byte-level representation.

Related Work. Side channel cryptanalysis was first suggested in [12] (cf. [13]). Template attacks were introduced in [5] and further explored in papers such as [24,20,7]. Algebraic side-channel attacks were introduced by Renauld et al. in [21,22], and first applied to the block ciphers PRESENT [3] and AES [15]. These works showed how keys can be recovered from a single measurement trace of these algorithms implemented in an 8-bit microcontroller, provided that the attacker can identify the Hamming weights of several intermediate computations during the encryption process. Already in these papers, it was observed that noise was the main limiting factor for algebraic attacks. To mitigate this issue, a heuristic solution was introduced in [22], and further elaborated in [28,14]. The main idea was to adapt the leakage model in order to trade some loss of information for more robustness, for example by grouping hard-to-distinguish Hamming weight values together into sets. An alternative proposal [17] suggested to include the imprecise Hamming weights in the equation set, and to deal with these imprecisions via the solver.

Despite their success, using generic SAT solvers or Pseudo-Boolean solvers still leaves room for improvement. The difficulties stem from the fact that in order to use them, the cipher representation has to be reduced to the bit-level. For byte-oriented ciphers this produces very large and complex instances, that are challenging to construct and debug. [16] notes that an AES equations instance may reach a size of 2.3 MB, depending on the methodology used to construct the equations. However, the most problematic aspect of bit-level solvers is their unpredictable, and often very long, run times. In [18] the authors report that run times vary over an order of magnitude between 8.2 hours to more than 143 hours on instances belonging to the same data set. The solver behavior is very sensitive to technical representation issues, and is controlled by a myriad of configuration parameters that are unrelated to the cryptographic task. Algebraic side-channel attacks which use local calculations were also considered in [26] and in [8].

Contribution. The focus of this work is a new *constraint solver*. Our solver embeds a model of the encryption process, accepts the known plain-text, and the output of the *decoder*, and outputs the highest probability keys with an estimation of their likelihood. However, unlike the algebraic attacks of [22] and [18], our constraint solver is not a general purpose Pseudo-Boolean or SAT-solver.

We wrote a special solver that is targeted at the unique types of constraints that occur in a side channel cryptanalysis of byte-oriented ciphers. Our solver is fundamentally probabilistic. It tracks the likelihoods of values in the secret key bytes, and updates them step by step through the encryption process, utilizing the probability distributions output by the decoder. A key ingredient in our framework is a novel method for reconciling multiple probability distributions for the same variable.

Applying our framework to a byte-oriented cipher with available side-channel information is quite natural and does not involve complex representation conversions into bit-level equations: the user needs to supply code snippets for the native byte-level operations of the cipher, arranged in a flow graph that embeds the functional dependence between the side channel leaks. Our framework uses a soft decision mechanism which overcomes realistic measurement noise and decoder classification errors.

As in previous solver-based attacks, our framework requires a *decoder*. The decoder accepts a single power trace, and outputs estimates of multiple intermediate values that are computed during the encryption and leaked by the side-channel. An estimate of a leaked value X in our framework is not a single "hard decision" value. Rather, as in [18], it is a probability distribution over the possible values of X. The decoder is usually constructed as a template decoder [5]. As in [18] we do not assume a Hamming-weight model for the leaked values - the decoder may output any probability distribution over the leak values. Note further that we do not impose a particular noise model on the decoder - e.g., it is not required to output only a single Hamming-weight value (or set of k values, as done by [28] and [18]).

We tested our framework on the DPA v4 contest dataset [2]. On this dataset, our framework is able to extract the correct key from one or two power traces with predictable and very short run times. Our results show a success rate of over 79% using just two measurements and typical run times are under 9 seconds. The source code can be downloaded from [27].

Organization. In the next section we introduce the probabilistic tools used in our solver. In Section 3 we describe the construction of the solver's flow graph. In Section 4 we show how we applied our method to AES. Section 5 includes the performance evaluation we conducted using the DPAv4 traces, and we conclude with Section 6.

2 Probabilistic Methodology

2.1 The Conflation Operator

A central part of our framework is a novel method of reconciling probability distributions. The basic scenario is as follows. Suppose we are trying to measure an unknown quantity X via two experiments. The outcome of the first experiment E_1 is a probability distribution P_{E_1} such that $P_{E_1}(X = i)$ is the likelihood

that X has value i. The second experiment E_2 measures the value of X using a different method, providing a second distribution P_{E_2}. We now wish to reconcile the results of these two experiments into a combined distribution $\hat{P}$. Intuitively, we want $\hat{P}$ to "strengthen" values on which E_1 and E_2 agree, and "weaken" values on which E_1 and E_2 differ. Thus, we want a probabilistic analogue to the logical "AND" operator. At one extreme, if $P_{E_1}(X = i) = 0$ (the value i is impossible according to E_1) then we want $\hat{P}(X = i) = 0$. At another extreme, if $P_{E_2}(X = i) = \frac{1}{N}$ for all N possible values of X (E_2 provides no information about X) then we want $\hat{P} = P_{E_1}$.

This general question was tackled by [9,10,11,6]. In particular, Hill [9] suggests a method called *conflation*, which is essentially the point-product of the distributions. In the case of two experiments E_1,E_2 the conflated probability $\hat{P} = \&(P_{E_1}, P_{E_2}) = (\hat{p}_1, .., \hat{p}_N)$ is defined as

$$\hat{p}_i = \hat{P}(X = i) = \frac{1}{\gamma} \cdot P_{E_1}(X = i) \cdot P_{E_2}(X = i)$$

where γ is a normalization factor to ensure $\sum_{i=1}^{N} \hat{p}_n = 1$. And in general, if multiple distributions $P^1, .., P^T$ are given then the conflated distribution is the normalized point product of all T distributions: $\hat{P} = \&(P^1, .., P^T) = (\hat{p}_1, .., \hat{p}_N)$ such that $\hat{p}_i = \frac{1}{\gamma} \prod_{t=1}^{T} p_i^t$

Hill [9] thoroughly analyzes the properties of the *conflation* operator. The paper shows that conflation is the unique probability distribution that minimizes the loss of Shannon Information. Further, conflation automatically gives more weight to more accurate experiments with smaller standard deviation. Finally, as desired, conflation with the uniform distribution is an identity transformation (i.e., it is indifferent to experiments with no information), and if $P^t(X = i) = 0$ for some i then $\hat{P}(X = i) = 0$ regardless of all other experiments. As we shall see, using conflation as the main probabilistic reconciliation method is extremely effective in our solver.

2.2 Conflating Probabilities of Single-Input Computation

In a byte-oriented cipher, many steps are transformations operating on a single byte. E.g., an XOR of a key byte X and a (known) plaintext byte is such a transformation. Similarly an SBox operation takes a single input X and produces $f(X)$. Suppose a template-based side channel oracle E_1 exists, that returns a probability distribution P_{E_1} of the values of X, and a second oracle E_2 returns a probability distribution P_{E_2} of the values of $f(X)$. Assuming the transformation $f(X)$ is deterministic and 1-1, then $P_{E_1}(X = a)$ should agree with $P_{E_2}(f(X) = f(a))$. Thus, we have two experiments measuring the value of $f(X)$: one is E_2, and the other is a permutation of the distribution E_1. Combining the experiment results via conflation gives us a more accurate distribution of $f(X)$ - and, equivalently, of values of X. Therefore, the reconciled probability for a single-input computation is defined to be:

$$\hat{P}(X = a) = \frac{1}{\gamma} P_{E_1}(X = a) \cdot P_{E_2}(f(X) = f(a)) \quad (1)$$

2.3 Conflating Probabilities of Dual-Input Computations

Suppose we have a function f of two independent byte values that outputs a byte: $f(X,Y) = Z$. We have oracles providing the probability distributions P_X, P_Y and P_Z for X, Y, Z respectively, and we wish to reconcile them. We first calculate the distribution P_f of $f(X,Y)$ based on P_X, P_Y: assuming X and Y are independent we get $P_f(c) = P(f(X,Y) = c) = \sum_{k,l:f(k,l)=c} P_X(k) \cdot P_Y(l)$. Now P_f and P_Z are distributions from two experiments estimating the same value Z, which we can conflate as before: $\hat{P} = \&(P_f, P_Z)$ so $\hat{P}(c) = P_f(c) \cdot P_Z(c) \cdot \frac{1}{\gamma}$ (for some normalization constant γ). However, we want to assign the reconciled probabilities $\hat{P}()$ to the inputs X and Y. Specifically, we want to split the probability $\hat{P}(c)$ among the pairs $(X = a, Y = b)$ for which $f(a,b) = c$ such that each pair will get its weighted share of $\hat{P}(c)$. Assume as before that $c = f(a,b)$, then the weighted split is:

$$\begin{aligned}\hat{P}(X=a,Y=b) = \hat{P}(c) \cdot \frac{P_X(a)\cdot P_Y(b)}{\sum_{k,l:f(k,l)=c} P_X(k)\cdot P_Y(l)} = \hat{P}(c) \cdot \frac{P_X(a)\cdot P_Y(b)}{P_f(c)} = \\ \frac{1}{\gamma} P_f(c) P_Z(c) \cdot \frac{P_X(a)\cdot P_Y(b)}{P_f(c)} = \frac{1}{\gamma} P_X(a) P_Y(b) P_Z(c)\end{aligned} \tag{2}$$

Thus we arrive at the following reconciled probability for the pair $X = a, Y = b$:

$$\hat{P}(X=a,Y=b) = \frac{1}{\gamma} P_X(a) P_Y(b) P_Z(f(a,b)) \tag{3}$$

3 Building Blocks

Our constraint model is a directed graph which describes the flow of information in the encryption process, as it affects the side channel leaks. The direction of the graph is from the unknown input bytes (the key in our case) to the output bytes (the ciphertext or intermediate values). Each part of the graph represents one of the following three constraint types: single-input constraint, dual-input constraint or data-redundancy constraint. There are two types of nodes in the graph:

1. Registry nodes - used to store possible values of intermediate values and their corresponding probabilities.
2. Compute nodes - used to connect registry nodes containing possible input values to registry nodes which should contain possible output values. Each compute node contains a code snippet implementing some step of the cipher.

3.1 Single-Input Computation Constraint

Suppose one of the steps of the cipher is a single-input byte function $f(X)$. Suppose we have two oracles, E_{in}, E_{out} providing the probability distributions of X and $f(X)$, respectively. Let $\alpha^{in}_{b_n} = P_{E_{in}}(X = b_n)$, and let $\alpha^{out}_{f(b_n)} = P_{E_{out}}(f(X) = f(b_n))$. These are the estimated probabilities of the input and output values given by the side channel information.

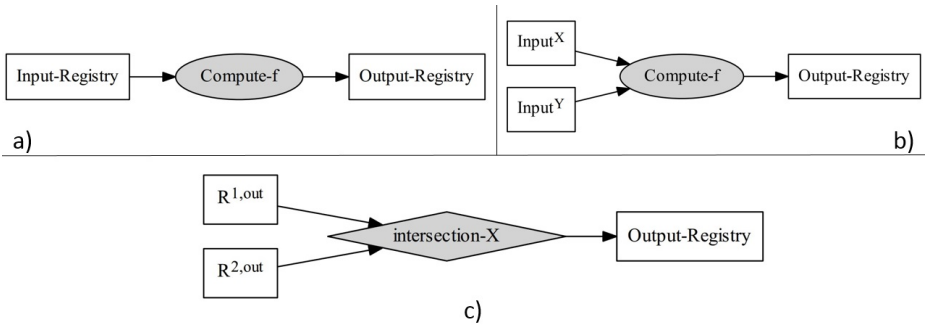

Fig. 1. Illustration of three types of constraints: a) single-input constraint, b) dual-input constraint, c) data-redundancy constraint

For a single input computation we define two registries: the *Input-Registry* contains the values $\{(b_n, \alpha^{in}_{b_n})\}$, and the *Output-Registry* contains the post-computation probabilities $\{(v_n, \alpha_{v_n})\}$ s.t $P(f(X) = v_n) = \alpha^{out}_{v_n}$.

We connect the input registry to the output registry via the *Compute-f* node (see Figure 1a), which contains a code snippet. The *Compute-f* node receives the tuples $\{(b_n, \alpha^{in}_{b_n})\}$ from the *Input-Registry*, computes the function f for each tuple, and for every value b_n outputs the tuple $(b_n, \alpha^{in}_{b_n}, f(b_n))$ to the *Output Registry*. Upon receiving the results from the compute function, the *Output-Registry* conflates $\alpha^{in}, \alpha^{out}$ as in Section 2.2: $\hat{\alpha}_n = \frac{1}{\gamma} P(X = b_n) \cdot P(f(X) = f(b_n)) = \alpha^{in}_{b_n} \cdot \alpha^{out}_{f(b_n)}$. After the computation is done the *Output-Registry* contains tuples of the form $(b_n, f(b_n), \hat{\alpha}_n)$.

3.2 Dual-Input Computation Constraint

Suppose a step in the cipher is a dual input byte-function $f(X, Y)$ such as an XOR of two intermediate values, and that side-channel information is available for $f(X, Y)$. In our constraint model we represent such a computation by two input registries entering a single compute node which includes the relevant code snippet (see Figure 1b). The compute node has to take into account all possible input combinations $\{b^X_n\} \times \{b^Y_n\}$. For every possible combination $(b^X_{n'}, b^Y_{n''})$ the compute node outputs the tuple $(b^X_{n'}, b^Y_{n''}, \alpha^{in,X}_{n'}, \alpha^{in,Y}_{n''}, f(b^X_{n'}, b^Y_{n''}))$. The output registry now needs to compute the conflated probability for the combination $(b^X_{n'}, b^Y_{n''}, f(b^X_{n'}, b^Y_{n''}))$. As described in Section 2.3, the conflated probability in the output registry is computed by

$$\hat{\alpha}_{n',n''} = \frac{1}{\gamma} \cdot \alpha^{in,X}_n \cdot \alpha^{in,Y}_n \cdot P(f = f(b^X_{n'}, b^Y_{n''}))$$

for a normalization factor γ.

3.3 Pruning Records from a Registry

The output size of a dual-input compute node is the product of sizes of the input registries. In some cases storing this much information is not feasible. For

example, when both input registries contain 256^2 records the output registry will have to hold 256^4 records, which is prohibitive. To avoid such a combinatorial explosion we can prune some of the records in the input registries by discarding all records with probabilities below a certain threshold t. Tuning the threshold is a trade off: selecting a tight threshold keeps combinatorial complexity low, but might cause pruning of records derived from the correct key bytes.

3.4 Data-Redundancy Constraint

We now deal with the case where some intermediate value X is used as input to more than one function. In our graph notation it means that some registry R^0 was used as input to two or more compute nodes, C^1, C^2. Denote the output registries of these compute nodes $R^{1,out}, R^{2,out}$. Each record in these registries contains the relevant value of X for that record. Enforcing a data-redundancy constraint over the value of X means that the records from $R^{1,out}, R^{2,out}$ should agree with each other probabilistically. For this purpose we introduce a special compute node which we call an *intersection* node (see Figure 1c). The records in $R^{1,out}, R^{2,out}$ are observations on the same value of X thus we can conflate their probabilities as before. Note that unlike the single-input or dual-input constraints, for an intersection node we do not require a side channel oracle. Note also that if the input-probability of some value is 0 then the conflated probability for that value remains 0. This means that if the registries entering an intersection node were pruned, the intersection node's output-registry only includes combinations of the un-pruned values.

3.5 Constructing a Solver for a Cipher

The structure of the solver's flow graph follows the information flow in the cipher, as reflected by the side channel leaks. At the beginning of the flow are the first unknown values - the key bytes. We now follow the cipher's first computation which is done on those key bytes, and construct the *compute nodes* which perform that computation with their code snippet. The compute node is connected to its input and output registries as in Section 3.1. We continue to chain single-input constraints until we reach a dual-input computation. We then use the dual-input constraint (Section 3.2) to describe this flow of information in the algorithm. In the registries used as inputs for a dual-input constraint we may wish to impose pruning to prevent a combinatorial explosion in the output registry. Note that each record in a registry contains all intermediate values used in the computation for the specific value in the record. Thus, different registries in the same layer may share some intermediate values. In that case, it is useful to combine these registries via a data-redundancy constraint. At the end of the flow we have registries containing values of intermediate computations. Each record has its assigned conflated probability and contains the key bytes values which led to this intermediate value, and the framework automatically does everything else.

Thus we see that in order to instantiate the framework for a specific cipher, we need to construct a flow graph that mimics the flow of data through the

cipher operations, with registries per side-channel leak. We need to supply code fragments for the compute nodes, select appropriate registries to prune and the pruning thresholds, and insert intersection nodes when possible.

4 Designing a Constraint Solver for AES

To evaluate our framework we built a constraint solver based on the side channel information from the first round of AES encryption, in a software implementation of the cipher. Our decoder extracted side channel information on:

1. 16 bytes of the output of AddRoundKey computation
2. 16 bytes of the output of SubBytes
3. 52 bytes from MixColumns computation:
   - 16 bytes of an XOR of 2 bytes, 4 in each column
   - 16 bytes of output of xtime computations , 4 in each column
   - 4 bytes of XOR of 4 bytes, 1 in each column
   - 16 bytes of output of the MixColumns computations

In total we have 84 intermediate byte values. For each leaked byte our decoder (see Section 5.2) produces a probability distribution over the 256 possible values.

Note that in the first round of AES the main diffusion operation is done by the MixColumns computation. MixColumns operates on groups of four bytes, thus a change of a single bit in the secret key can not affect more than four bytes of output (in the first round). This leads our constraint model to be a graph that can be divided into four connected components. Each connected component describes a constraint model for a single column. Each of the four components reflects the byte reordering done by the ShiftRows sub-rounds. This observation means that our solver actually works independently on each set of 4 key bytes.

4.1 Initialization and Single Input Computations

At the beginning of the computation for every key byte we consider all 256 values as possible. Since initially we do not have side channel information on the key bytes the probability for every value is $1/256$. The AddRoundKey and SubBytes sub-rounds are single input computation. Note that no computation is done in the ShiftRows sub-round, thus it does not leak additional information and is not used in our constraint model. The left side of Figure 2 illustrates the single-input constraints for four key bytes.

4.2 Basic Computation of MixColumns

A common implementation of the MixColumns computation in software on an 8-bit microcontroller (cf. [23]) is to compute the following intermediate values:

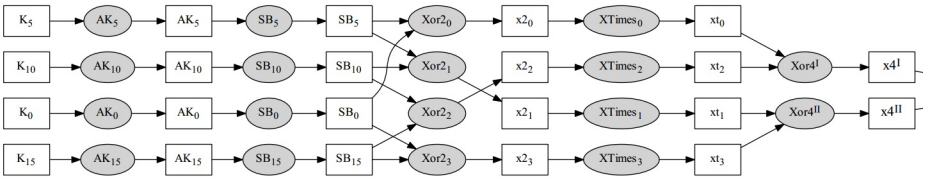

Fig. 2. Visual representation of the constraint solver tracking four key bytes up to the X4 computation in AES. Registry nodes are drawn as rectangles and compute nodes as ellipses. Abbreviations: AK-AddKey, SB-SubBytes

1. The XOR value of four column bytes:

$$x4 \leftarrow b_0 \oplus b_1 \oplus b_2 \oplus b_3$$

2. The XOR values of adjacent bytes:

$$\begin{aligned} x2_0 &\leftarrow b_0 \oplus b_1 \\ x2_1 &\leftarrow b_1 \oplus b_2 \\ x2_2 &\leftarrow b_2 \oplus b_3 \\ x2_3 &\leftarrow b_3 \oplus b_0 \end{aligned}$$

3. The multiplication by 2 in Galois field $\mathbb{F}_{2^8}$ ("xtime") of the four values above:

$$xt_i \leftarrow 2 \cdot x2_i \mid_{\mathbb{F}_{2^8}} \text{ for } 0 \leq i \leq 3$$

Constructing the $x2_i$ constraints is done by using 4 dual-input compute nodes followed by a single-input constraint, for xtime (see Figure 2).

4.3 Pruning

Until the $x2_i$ registry, the AddKey and SubBytes registries contain 256 records for each of the 256 possible key bytes. Thus, the $x2_i$ registries and hence xt_i registries contain 256^2 records each. If we naively use the xt_i registries as input for a dual-input constraint $X4$ to compute the XOR of four values - it means that $x4$ registry will contain 256^4 records, which is prohibitive. We note that by the time we reach the xt_i registry the probability assigned to each record is conflated over 6 side channel leaks: 2 AddRoundKey bytes, 2 SubBytes bytes, a single x2 byte and a single xtime byte. Therefore, the conflated probabilities of incorrect key bytes have dropped significantly. Hence, this is a good spot in our constraint model to perform pruning. We chose to prune all records with probability of less than $t = 10^{-25}$. This specific value keeps the correct records for 92% of the 600 traces we experimented with. On the other hand, this t value leaves no more that 500 records (out of 65536) in each xt_i registry, leading to low memory consumption and fast running times

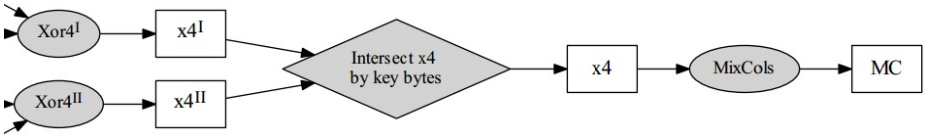

Fig. 3. Visual representation of the constraint solver tracking four key bytes, of column 0, from $x4$ to MixColumns computation. MC stands for MixColumns

4.4 Computing the Output of MixColumns

Each record in the xt_i registry contains all the values involved in the computation path. That is: 2 plaintext bytes, 2 key bytes, 2 AddRoundKey bytes, 2 SubBytes output values, 1 value of XOR of 2 bytes and 1 value of the *xtime* operation on that XOR output. Here we can make a useful observation: We have leaks for $x4$ and also for $x2_0, x2_1, x2_2, x2_3$. But these leaked values need to be self-consistent regardless of how the implementation actually computes $x4$:

$$x4^I = x2_0 \oplus x2_2$$
$$x4^{II} = x2_1 \oplus x2_3$$

Thus we can compute (and conflate) the values of $x4$ in two ways. Since the xt_i registries contain the corresponding values of $x2_i$ we can use these registries as inputs for two parallel dual-input Compute-$x4$ nodes. Figure 2 illustrates the constraint solver up to the $x4^I$, $x4^{II}$ registries.

Assuming we did not prune the records of the correct combination of key bytes, the quartet of the correct key bytes should appear in records of both $x4^I$ and $x4^{II}$ registries. Thus we now use a data-redundancy constraint (recall Section 3.4) to intersect records according to the 4 key bytes. The output of the data-redundancy node is inserted into a registry called $x4$. Each record of that registry contains all the byte values used for that specific record, that is: 4 plaintext bytes, 4 key bytes, 4 SubBytes outputs, 4 outputs of XOR of 2, 4 outputs of xtime computations, and 1 value of XOR of 4.

Each record in the $x4$ registry contains all the information required to compute the 4 output bytes of MixColumns. Since we use a single record to compute a tuple of 4 output bytes - we consider this computation as a single-input computation. As before let $\{\alpha^{in}\}$ denote the conflated probabilities of records in $x4$ registry. Since MixColumns has 4 output bytes - we have four leaks to conflate with, representing the separate side channel information on the four output bytes: $\{\alpha^{out,0}\}$, $\{\alpha^{out,1}\}$, $\{\alpha^{out,2}\}$, $\{\alpha^{out,3}\}$. The conflated probability is given by: $\hat{\alpha} = \alpha^{in} \cdot \alpha^{out,0} \cdot \alpha^{out,1} \cdot \alpha^{out,2} \cdot \alpha^{out,3}$. $\hat{\alpha}$ is then normalized so that all probabilities sum to 1. The final result is the MC registry. Figure 3 illustrates the constraint solver from $x4^I$, $x4^{II}$ registries to the MC registry.

4.5 Finding the Keys

We now have in each MC registry, for each "column", a set of records representing the possible computation paths and their corresponding probabilities. Recall that

a "column" is defined at the entrance to MixColumns, so the key byte indices are reordered by the ShiftRows operation. Each registry record represents a candidate combination of 4 key bytes. Together all the MC registries contain possible combinations of 16 key bytes.

A naive way to iterate over the key candidates would be to sort the registries in decreasing probability order, to set some upper bound R, and to try all candidates from ranks r_1, r_2, r_3, r_4 s.t. $r_i \leq R$ (one per MC registry). This approach is bounded by R^4 key tries. However, using the method of [25], it is possible to iterate over these R^4 keys according to their probabilities, thus speeding up the key search. An alternative method for reducing the candidate keys is to run the constraint solver twice using different power traces and then intersect the groups of key candidates.

5 Performance Evaluation

5.1 Experimental Setup

We instantiated our framework for AES, and executed it on power traces extracted from a real implementation of an AES-256 variant. The implementation is the one presented in the DPA contest v4 [2]. This implementation contains a power-analysis counter measure called RSM described in [1]. The deviations from the classic AES are:

1. RSM-AES utilizes an arbitrary fixed 16-byte *Mask*. At the beginning of the encryption process a random *offset* between 0 to 15 is drawn. Let o denote the *offset*, and let m^o denote the cyclic rotation of *Mask* by offset o.
2. The 16 bytes of plaintext are XOR-ed with m^o. Let pm be the result, i.e., $pm_i = p_i \oplus m_i^o$, $0 \leq i \leq 15$.
3. In the AddRoundKey sub-round the round key is XOR-ed with pm instead of the plaintext.
4. RSM-AES uses different S-BOXs for every byte, which are derived from the value of the m^o.
5. The ShiftRows and MixColumns sub-rounds are unchanged.
6. An additional sub-round is added to extract the unmasked cipher text, but it is not relevant in our attack since the power traces only cover the first round.

5.2 Decoding

To profile the power consumption behavior of the RSM-AES implementation we used techniques similar to those of [19]. Our leak model is the Hamming-weight model. This model was chosen since our experiments showed high correlation with the Hamming weights of the intermediate values. Using the raw values, on the other hand, showed very low correlation. 100 classifiers were trained to classify the Hamming weights of 100 intermediate values. Of these, 84 intermediate values are those described in Section 4, and 16 values are the masked plaintext bytes of the RSM counter-measure (see pm_i description in Section 5.1). We used

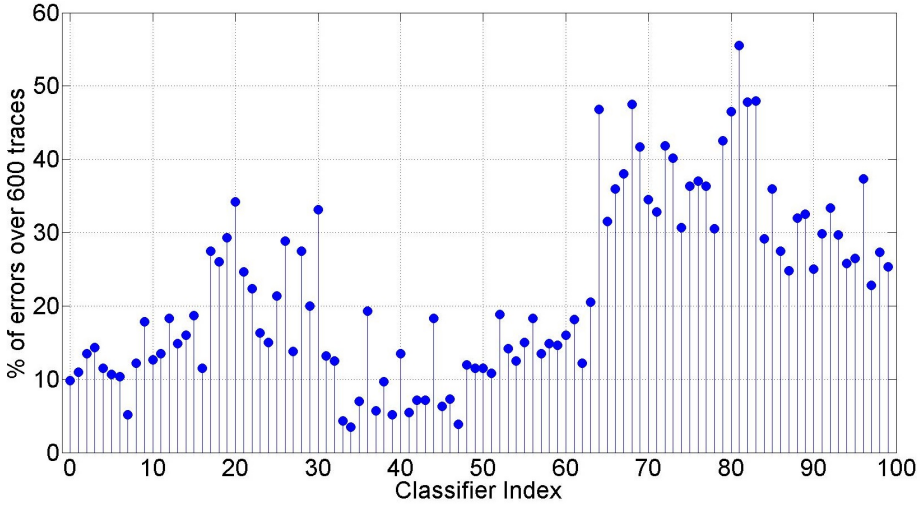

Fig. 4. Percent of classification errors per classifier, evaluated over 600 traces. Classifiers 0-15 are for Hamming-weights of pm_i, 16-31 are for AddKey outputs, 32-47 are for SubBytes, 48-63 are for $x2$, 64-79 are for xt, 80-83 are for $x4$ and 84-99 are for MixColumns outputs.

200 traces to train the classifiers and an additional 200 traces to evaluate the classifiers' performance, in order to select the best trace-samples to be used as inputs for each classifier. As described in [19] the classifiers were trained to identify Hamming weights 2-6 and were then extrapolated to classify all 9 possible Hamming weight values 0-8.

Let C_l be the classifier for leak l, and let $C_l(hw)$ be the probability that classifier C_l assigns to the event that the correct value has a Hamming-weight of hw for $hw \in 0..8$. To evaluate the classifiers performance, we define a classification error to be when the Hamming-weight with the highest probability, as predicted by the classifier, is not the correct Hamming-weight. Our decoder is far from perfect: most classifiers have an average error rate of 10-20% and some have an error rate as poor as 55%. Some intermediate values are decoded with low error rates (e.g., SubBytes) while others are harder to decode (e.g., MixColumns). Specific classifiers' error rates are shown in Figure 4.

Note that in our framework a classifier failing to predict the exact Hamming-weight as the most likely value still conveys significant information: as long as the correct Hamming-weight has higher probability than other incorrect Hamming-weight classes, it helps the solver distinguish the correct values from the incorrect ones. As we will see, even with these far-from-perfect classifiers, our framework is able to find the correct keys.

5.2.1 Overcoming the RSM Counter Measure. As described above, we have 16 classifiers C_i, $0 \leq i \leq 15$, trained to estimate the probabilities of the

Hamming-weight values of $pm_i = p_i \oplus m_i^o$, where m_i^o is the i^{th} byte of the Mask rotated by offset o. For every possible value of $o \in 0..15$ we derive 16 mask bytes m_i^o and compute $pm_i^o = p_i \oplus m_i^o$. Let $HW(x)$ denote the Hamming weight of x. Recall that for a given value hw, $C_i(hw)$ is the probability estimation of the decoder C_i of $HW(pm_i)$, i.e $C_i(hw) = P(HW(pm_i) = hw)$. For every value of o, we compute the offset score: $S(o) = \prod_{i=0}^{15} C_i(HW(pm_i^o))$. The *offset* o which gave the highest score $S(o)$ is declared the correct one. We experimented with this method on 600 traces (distinct from the 400 training traces) and measured an offset prediction success rate of 100%. Thus we see that the 4-bit side-channel counter-measure used in RSM-AES offers no protection against template based attacks, even without a constraint solver.

5.2.2 Probability Estimation for 256 Values. Our constraint solver uses a soft-decision decoder: it requires as input a probability estimation for 256 possible values of every intermediate computation. We do not filter out the less likely Hamming weights: instead we split the 9-value distribution given by C_l among the byte values X, according to their Hamming-weights. Let

$$S_{hw} = \|\{x \in \{0..255\} | HW(x) = hw\}\| \text{ for } 0 \leq hw \leq 8$$

be the number of values between 0-255 with Hamming weight hw. For every intermediate byte value b_l among the 84 leaks $l \in 0..83$ and classier C_l - we assign the probability for value $x \in 0..255$ to be $P(b_l = x) = \frac{C_l(HW(x))}{S_{HW(x)}}$. Note that $\sum_{x=0}^{255} P(b_l = x) = 1$.

5.3 Implementation of the Constraint Solver

The custom solver designed for AES as described in Sections 3 and 4 was implemented in Matlab R2013a. Our code consists of 6200 lines of code over 25 files. The implementation consists of general *registry* and *compute* blocks, and specialized compute classes to be used by the general compute blocks. Other than the 4 registries used for the intersection constraints, each registry is associated with a specific leak l among the 84 leaks. They therefore receive an a-priori probability estimation for every value X as explained in Section 5.2.2. These are the α^{out} values described in Section 3.1. The graph representing the full constraint solver is depicted in Figure 7.

5.4 Results and Discussion

We ran our solver on an Intel core i7 2.0 GHz PC running Ubuntu 13.04 64 bit, with 8 GB of RAM and a SSD hard drive. Over 600 traces the median running time of decoding + running the solver was 9 seconds. Solving of 98% of the experiments completed in under 30 seconds. The maximum running time was 85 seconds.

At the end of a run, each of the four MixColumns output registries contains records with 4-key-byte candidates. A full 16 byte key is constructed by taking

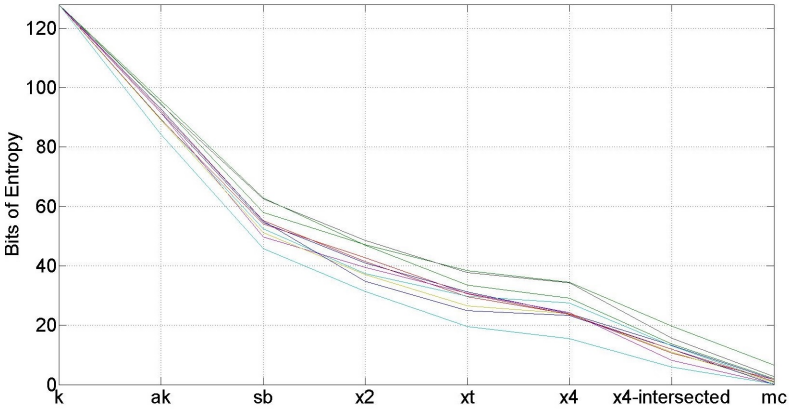

Fig. 5. Evolution of entropy of 16 key bytes at different solver phases, for 10 runs on randomly selected traces. Abbreviations: ak - AddKey, sb - SubBytes, x2 - XOR of 2 bytes, xt - xtime, x4 - XOR of 4 bytes, mc - MixColumns.

a record from each of the four MixColumns registries. The median number of 4-key-byte candidates (for a single column) was 43930, and the median number of full key candidates was $2^{61.2}$. To measure the solver's success, for each registry we look at the rank of the record containing the correct 4-key-byte combination. If the maximum rank of the correct key quartets in all four registries is lower than R, then exhaustive search for the correct key would require no more that R^4 tries. We found that in 38% out of 600 power traces, at least 3 key quartets were among the top 5 records. The correct key in over 50% of the traces can be found in less then $R^4 = 2^{30}$ attempts. We believe that using the optimized algorithm of [25] to iterate over key candidates according to probability would significantly decrease the number of tries before finding the correct key. We did not test the approach of [25] on our results. Instead, we opted to use a second power trace and intersect the candidate key-quartets (see below).

Figure 5 shows how the Shanon entropy of 16 key-bytes drops as the solver uses the side channel leaks. At the beginning of the flow each key byte has probability of $\frac{1}{256}$, giving $Entropy = 128$, as expected for 128 unknown bits of key. Figure 5 shows that the entropy dropped from 128 down to 0.2-6.6 bits. This means that although the solver outputs a median of $2^{61.2}$ key candidates, the probability mass is concentrated over very few candidates.

When more than one power trace is available for the attack, we can run the decoding + solver on each trace and intersect the candidate keys. The intersection is done separately on the 4-key-byte candidates for every column, and the probability distributions are conflated. To measure the performance of this approach we ran 250 experiments, each with independent traces. When the intersection was not empty, the median number of candidates per column was 4 and the median number of full key candidates was 315. Figure 6 shows how many power traces were required to yield the correct key as the first ranked

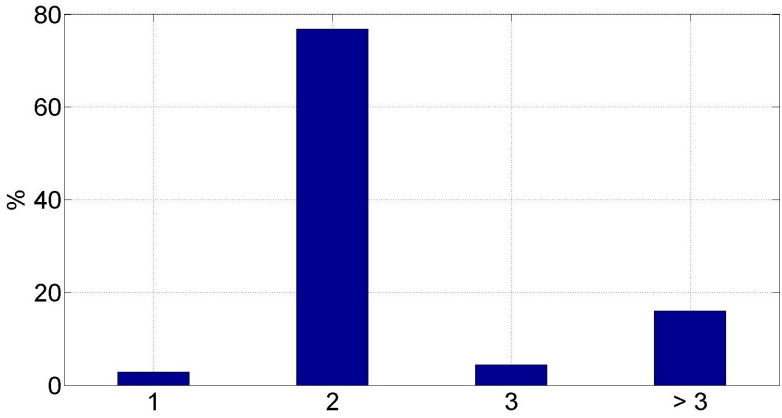

Fig. 6. Number of power traces needed to find the correct key.

candidate. It shows that with only 2 traces we can identify the correct key as the top candidate with success rate of 79.6%.

We submitted our solver to the DPA v4 contest. The formal evaluation process of the DPA contest is equivalent to a single experiment (in contrast to the 250 we performed). According to the above statistics, a single experiment has 20.4% chance of needing more than 2 traces - as actually happened. When more than 2 traces are used, our solver requires more time to perform the intersection between the possible key candidates. The DPA v4 hall of fame lists our contribution as requiring 5 traces and 55 seconds per trace to complete. As of the date of writing, our solver is one of the leading entries in the contest.

6 Conclusions and Future Work

In this paper we described a specialized byte-oriented constraint solver for side channel cryptanalysis. Instead of representing the cipher as a complex and unwieldy set of bit-level equations, the user only needs to supply code snippets for the native operations of the cipher, arranged in a flow graph that models the dependence between the side channel leaks. Through extensive use of the conflation technique our solver is able to reconcile low-accuracy and noisy measurements into an accurate low-entropy probability distribution, with extremely low and very predictable run times. On the DPA v4 contest dataset our framework is able to extract the correct key from one or two power traces in under 9 seconds with a success rate of over 79%.

The technique is not dependent on the decoding method, does not assume a Hamming-weight model for the side channel, and does not impose any particular noise model. It can be applied as long as it is possible to decode the side-channel trace into a collection of probability distributions for the intermediate values. We believe it would be quite interesting to test our framework against other

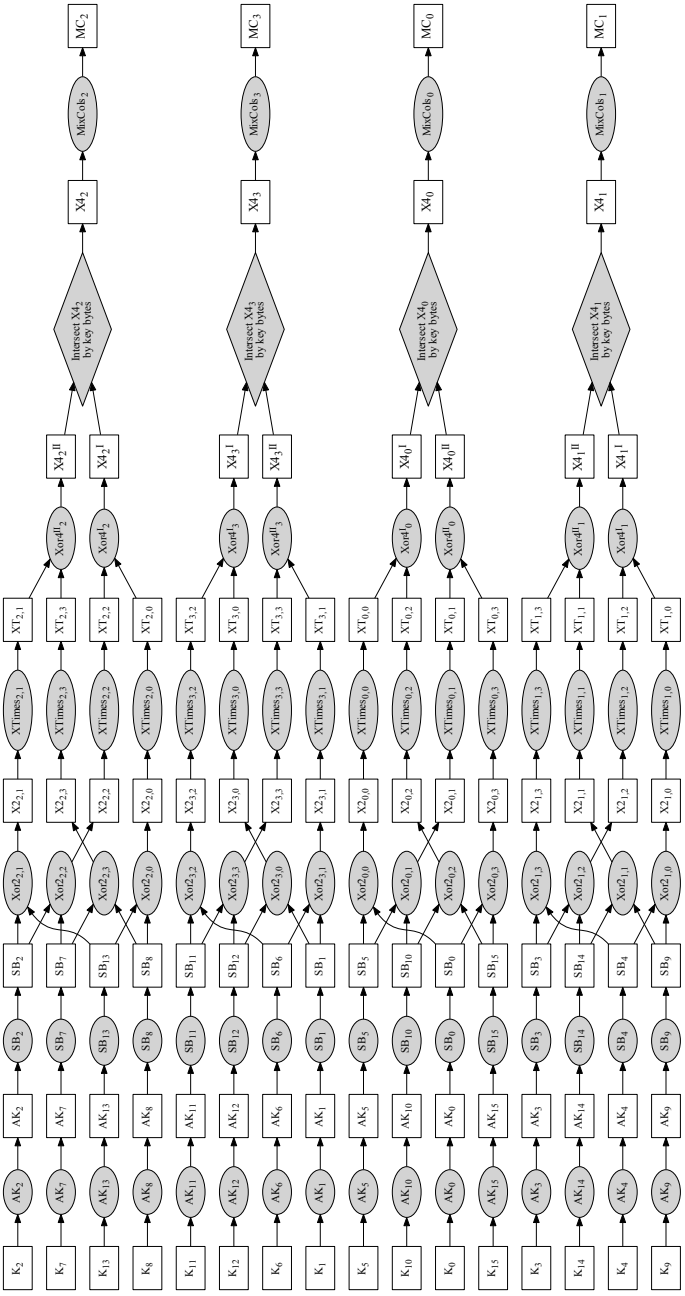

Fig. 7. Flow Graph of the Full AES Constraint Solver

implementations of AES, against other types of side-channel information, and against other byte-oriented ciphers.

References

1. Description of the masked AES of the DPA contest v4, http://www.dpacontest.org/v4/data/rsm/aes-rsm.pdf
2. DPA contest v4, http://www.dpacontest.org/v4/
3. Bogdanov, A.A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C.: PRESENT: An ultra-lightweight block cipher. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 450–466. Springer, Heidelberg (2007)
4. Brier, E., Clavier, C., Olivier, F.: Correlation power analysis with a leakage model. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 16–29. Springer, Heidelberg (2004)
5. Chari, S., Rao, J.R., Rohatgi, P.: Template attacks. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 13–28. Springer, Heidelberg (2003)
6. Clemen, R.T., Winkler, R.L.: Combining probability distributions from experts in risk analysis. Risk Analysis 19(2), 187–203 (1999)
7. Elaabid, M.A., Guilley, S.: Practical improvements of profiled side-channel attacks on a hardware crypto-accelerator. In: Bernstein, D.J., Lange, T. (eds.) AFRICACRYPT 2010. LNCS, vol. 6055, pp. 243–260. Springer, Heidelberg (2010)
8. Guo, S., Zhao, X., Zhang, F., Wang, T., Shi, Z.J., Standaert, F., Ma, C.: Exploiting the incomplete diffusion feature: A specialized analytical side-channel attack against the aes and its application to microcontroller implementations. IEEE Transactions on Information Forensics and Security 9(6), 999–1014 (2014)
9. Hill, T.: Conflations of probability distributions. Transactions of the American Mathematical Society 363(6), 3351–3372 (2011)
10. Hinton, G.E.: Training products of experts by minimizing contrastive divergence. Neural Computation 14(8), 1771–1800 (2002)
11. Kahn, J.M.: A generative bayesian model for aggregating experts' probabilities. In: Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence, pp. 301–308. AUAI Press (2004)
12. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
13. Mangard, S., Oswald, E., Popp, T.: Power Analysis Attacks: Revealing the Secrets of Smart Cards (Advances in Information Security). Springer-Verlag New York, Inc., Secaucus (2007)
14. Mohamed, M.S.E., Bulygin, S., Zohner, M., Heuser, A., Walter, M., Buchmann, J.: Improved algebraic side-channel attack on AES. Journal of Cryptographic Engineering 3(3), 139–156 (2013)
15. Information Technology Laboratory (National Institute of Standards and Technology). Announcing the Advanced Encryption Standard (AES). Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology Gaithersburg, MD (2001)
16. Oren, Y.: Secure hardware - physical attacks and countermeasures. PhD thesis, Tel-Aviv University, Isreal (2013), https://www.iacr.org/phds/?p=detail&entry=893

17. Oren, Y., Kirschbaum, M., Popp, T., Wool, A.: Algebraic side-channel analysis in the presence of errors. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 428–442. Springer, Heidelberg (2010)
18. Oren, Y., Renauld, M., Standaert, F.-X., Wool, A.: Algebraic side-channel attacks beyond the hamming weight leakage model. In: Prouff, E., Schaumont, P. (eds.) CHES 2012. LNCS, vol. 7428, pp. 140–154. Springer, Heidelberg (2012)
19. Oren, Y., Weisse, O., Wool, A.: Practical template-algebraic side channel attacks with extremely low data complexity. In: Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy, p. 7. ACM (2013)
20. Oswald, D., Paar, C.: Improving side-channel analysis with optimal linear transforms. In: Mangard, S. (ed.) CARDIS 2012. LNCS, vol. 7771, pp. 219–233. Springer, Heidelberg (2013)
21. Renauld, M., Standaert, F.-X.: Algebraic side-channel attacks. In: Bao, F., Yung, M., Lin, D., Jing, J. (eds.) Inscrypt 2009. LNCS, vol. 6151, pp. 393–410. Springer, Heidelberg (2010)
22. Renauld, M., Standaert, F.-X., Veyrat-Charvillon, N.: Algebraic side-channel attacks on the AES: Why time also matters in DPA. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 97–111. Springer, Heidelberg (2009)
23. Stallings, W.: Cryptography and Network Security ch. 5, 6th edn. Pearson (2014)
24. Sugawara, T., Homma, N., Aoki, T., Satoh, A.: Profiling attack using multivariate regression analysis. IEICE Electronics Express 7(15), 1139–1144 (2010)
25. Veyrat-Charvillon, N., Gérard, B., Renauld, M., Standaert, F.-X.: An optimal key enumeration algorithm and its application to side-channel attacks. In: Knudsen, L.R., Wu, H. (eds.) SAC 2012. LNCS, vol. 7707, pp. 390–406. Springer, Heidelberg (2013)
26. Veyrat-Charvillon, N., Gérard, B., Standaert, F.-X.: Soft analytical side-channel attacks. Cryptology ePrint Archive, Report 2014/410 (2014), `http://eprint.iacr.org/2014/410`
27. Weisse, O.: Source code of our constraint solver as submitted to DPA v4 contest, `http://www.ofirweisse.com` see DPA v4
28. Zhao, X., Wang, T., Guo, S., Zhang, F., Shi, Z., Liu, H., Wu, K.: SAT based error tolerant algebraic side-channel attacks. In: 2011 Conference on Cryptographic Algorithms and Cryptographic Chips, CASC (2011)