The Raymond and Beverly Sackler Faculty of Exact Sciences

The Blavatnik School of Computer Science

New Methods for Side Channel Cryptanalysis

Thesis submitted in partial fulfilment of the requirements towards the M.Sc. degree

by

Ofir Weisse

This work was carried out under the supervision of

Prof. Avishai Wool

and

Dr. Eran Tromer

October 2014

Acknowledgements

I would like to thank my advisor, Prof. Avishai Wool. His professional help, patience and superb guidance were vital to the success of this research. I would also like to thank Dr. Yossef Oren for introducing me to the practical aspects of side channel cryptanalysis. His mentoring as well as his encouragement for creativity were an enormous help throughout the research.

Abstract

Template-based Tolerant Algebraic Side Channel Attacks (Template-TASCA) were suggested by Wool et al. in 2012. as a way of reducing the high data complexity of template attacks by coupling them with algebraic side-channel attacks. In contrast to the maximum-likelihood method used in a standard template attack, the template-algebraic attack method uses a constraint solver to find the optimal state correlated to the measured side-channel leakage. In this work we present a practical application of the template-algebraic key recovery attack to a publicly available data set (IAIK WS2). We show how our attack can successfully recover the encryption key even when the attacker has extremely limited access to the device under test – only 200 traces in the offline phase and as little as a single trace in the online phase. However, to use such solvers 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 work 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%.

Contents

1 Introduction

Side-channel attacks are attacks which reveal the secrets of cryptographic devices by observing their physical properties [16]. 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:

1. *Profiling*: Profiling is an offline activity. a reference device which is similar to the device under test (DUT) but entirely under the control of the attacker is profiled and characterized. 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 [7]. 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 [28], but many other decoders are possible. An effective profiling method is the template attack, which was introduced in [9]. The offline profiling phase thus outputs a series of **decoders**, each of which maps a certain set of interesting points in the trace to a certain set of secret values.

2. *Decoding*: After the profiling phase, in the online phase, the attacker is provided with a small number of power traces (typically, a single trace). The decoders created in the profiling phase are applied 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.

3. *Solving*: Let k be an encryption key and p be a plaintext. Then the vector $s_{1\cdots m} = State\,(k, p)$ is a description of the internal state progression experienced by the DUT as it encrypts the plaintext p under the key k. Similarly to the encryption key, the encryption state is also divided into elements such as bytes. In contrast to the key bytes, which are usually chosen independently at random during the key generation process, each byte of the state is typically dependent on other bytes, and not all combinations of state bytes correspond to the transcript of a valid encryption. For example, fixing the values of the encryption key and the plaintext completely determines the values of the entire encryption state.

It is well known that the side-channel trace contains information not only about the key bits themselves, but also about the state bits. This property is used to great effect by correlation power analysis attacks [8] and similar side-channel attacks. The objective of the **solving** step is to choose the most likely **state** instead of the most likely **key**. Since more information is extracted from the trace, and since this state information contains some internal redundancy, it is reasonable to assume that such an attack would be able to tolerate a lower level of accuracy in the recovery of individual state elements. Reducing the accuracy requirement for the individual templates in turn translates to reduced data demands both for the offline and the online phase of the attack.

Thus in the **solving** step 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. Such possible solvers are: a SAT-solver [27, 28, 34] or a Pseudo-Boolean solver [22]. The solver solves the problem instance, outputting the best key candidates satisfying the constraints.

1.1 Related Work and Motivation

Side channel cryptanalysis was first suggested in [15] (cf. [16]). Template attacks were introduced in [9] and further explored in papers such as [11, 25, 30]. Algebraic side-channel attacks were introduced by Renauld et al. in [27, 28], and first applied to the block ciphers PRESENT [6] and AES [19]. 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.

The first challenge in performing such attacks is to be able to distinguish between 9 classes of Hamming weights: note that Hamming weight of value 0,1,7,8 are relatively rare: only 0 maps to Hamming weight 0, only 255 maps to Hamming weight 8, and there are only eight values for Hamming weight 1 and eight values for Hamming weight 7. This requires the attacker to have a large training set in the **profiling** phase in order to properly profile the power traces with leaks corresponding to those Hamming weight values.

Already in the above papers, it was observed that noise was the main limiting factor

for algebraic attacks. To mitigate this issue, a heuristic solution was introduced in [28], and further elaborated in [18, 34]. 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 [21] 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. [20] 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 [22] 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.

1.2 Contribution

This work presents advances in two subjects: **profiling** and **solving**.

1.2.1 Advances in Profiling

Firstly, this work presents a practical profiling method which requires a relatively small dataset of power traces. This work shows an evaluation of the template-algebraic side-channel attack. The described profiling attack can find the optimal key based on the a-posteriori probabilities of the entire state of the DUT, and not just those of the individual key bytes. We describe the template-algebraic approach and show how it can be used to effectively recover the secret key with an extremely reduced data complexity, both at the offline and online phases, when compared to standard template attacks. We evaluate the attack on a public data set of real-world power traces, and show how it can recover an AES secret key with a very high success rate, even when the offline profiling phase is provided with less than 200 traces and the online attack phase is provided with one or two traces. The median running time of our attack is 600 seconds for a

two-trace scenario, and 25 hours for a single-trace scenario. The advance in profiling appeared in [23] and is described in chapters 2-5.

1.2.2 Advances in Solving

Secondly, this work presents a new *constraint solver* for the **solving** step. Our solver embeds a model of the encryption process, accepts the known plaintext, and the output of the *decoder*, and outputs the highest probability keys with an estimation of their likelihood. However, unlike the algebraic attacks of [28] and [22], our constraint solver is not a general purpose Pseudo-Boolean or SAT-solver.

We wrote a special solver that is focused 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 [22], it is a probability distribution over the possible values of X. The decoder is usually constructed as a template decoder [9]. As in [22] 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 [34] and [22]).

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 [24]. The new solver is described in [24] and in chapters 6-9.

2 Profiling

Our goal in this phase is to generate a decoder that is given a single power trace x and can produce leakage information about the encryption intermediate states. The leakage information consists of the a-posteriori probabilities $Pr(s_i|x)$. In the profiling phase we learn the power consumption behavior of the *Device Under Test* (DUT). To this end we assume that we have the exact same model of the DUT, and that we are able to run it many times with known plaintext and keys. Later, in the online phase, we use the template to extract the a-posteriori probabilities for the intermediate state bytes from the DUT.

2.1 Leaks of Information

As stated in [16], micro-controller implementations of AES are expected to leak the Hamming weights of the state bytes they process. The leaks used for this work are:

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 leaks from 84 intermediate byte values.

Denote the intermediate state byte for a certain leak i as s_i, and the Hamming weight of that state byte by $HW(s_i)$. As described in [16], we assume that the Hamming weight leakage of information in the power consumption is of the form

$$Power = Power\left(HW\left(s_i\right)\right) + noise.$$

noise is additional noise due to other calculations performed by the controller and thermal noise. This noise is assumed to be normally distributed, but with unknown parameters. Such characteristics are exactly the probabilistic model used to construct a Bayesian classifier.

2.2 Methodology

In the following discussion the term *trace* represents a vector of *samples*. Each sample represents the power consumption of the DUT at a certain point in time. Our data set is taken from the on-line material of the book [16] (see URL in reference), under "workspace 2", also called WS2. In the WS2 data set we are given 200 traces, each of which consists of 100,000 samples and corresponds to the execution of a full AES round. A *trace-step* represents an index into a trace (a value between 1-100,000) representing a point in time, and a *trace-distance* represents the difference between two trace steps. *Features* are *samples* we select, across all *traces*, to be used as input for the classification algorithm.

The profiling step consisted of three general stages: First, we created a set of candidate classifiers, each of which operates on a small amount of features. Next, we applied a greedy algorithm to combine the best features used by the candidate classifiers and create a single classifier which operates on a large amount of features. Finally, we used knowledge about the physical model of the device under attack to optimize the decoding performance of our classifier. Each step is explained in more detail below.

We built 2 variants of the decoder. Variant I was used with WS2 dataset and evaluated with an algebraic solver using similar methods of [21, 22]. See Sections 3,4. Variant II was an improved decoder. It uses a different scoring method for evaluating the amount of information each sample holds, and evaluates classifiers' performance in a different way. Variant II was evaluated with our new proposed solver introduced in Section 6.

2.2.1 Finding Candidate Features

The first step of building the decoder was identifying the points in time where the traces contain information about an intermediate value or operation which is of interest to the attacker. Given the 200 traces, we would like to learn how the power consumption behaves in each of the Hamming weight cases 0 through 8. For each leak (among the

84, see Section 2.1) we divide the 200 traces into 9 Hamming weight classes (0 through 8) according to the true leak takes.

Notice that Hamming weights of value 0,1,7,8 are relatively rare: only 0 maps to Hamming weight 0, only 255 maps to Hamming weight 8, and there are eight values for Hamming weight 1 and eight values for Hamming weight 7. This fact, combined with the small size of our training set, means that there are only a few, if any, traces with leaks corresponding to those Hamming weight values. For this reason we merge classes 0 and 1 into class 2, and classes 8 and 7 into class 6.

After dividing our 200 traces into the 5 classes (2 through 6), we need to build a classifier that can distinguish between traces of these classes for the current leak. Similar to what was done in [26], we use a naive Bayes classifier. In this type of classifier each class is represented by a mean value μ and variance σ. In our case, we chose to train each classifier on multiple samples taken from several different trace-steps, across all traces. This makes our classifier multidimensional, with the number of dimensions equal to the number of features we selected. Thus, μ and σ values are actually vectors of dimension equal to the number of features per class.

One important element in training a good classifier is selecting the right input features from a very large input data. Since each trace consists of 100,000 samples, there are 100,000 possible features for each trace. We wanted to find candidate features in the traces, for every leak among the 84 leaks. We follow the recommendation of [26]: instead of taking sequential samples from a trace as features, we took samples with a certain minimal trace-distance of n from each other, where n is the number of trace-steps corresponding to a measurement of a full clock cycle of the DUT. After evaluating different values for n, we observed a significant improvement in classification results when training our classifier on triplets of samples with distances of 8 trace-steps between them.

To select candidate features for our classifier, we performed an iterative feature ranking. For $0 \leq i \leq 100,000$ we try to train a classifier on 200 traces with input features taken as the samples $traces(i)$, $traces(i + 8)$, $traces(i + 16)$. We call this classifier the i^{th} classifier. To quickly measure the performance of this three-featured classifier we perform a 4-fold cross validation: we split our 200 traces into four groups, containing 50 traces each; trained over $3 \cdot 50 = 150$ traces and tested performance on the remaining 50 traces. Repeat this process another 3 times so each group will be

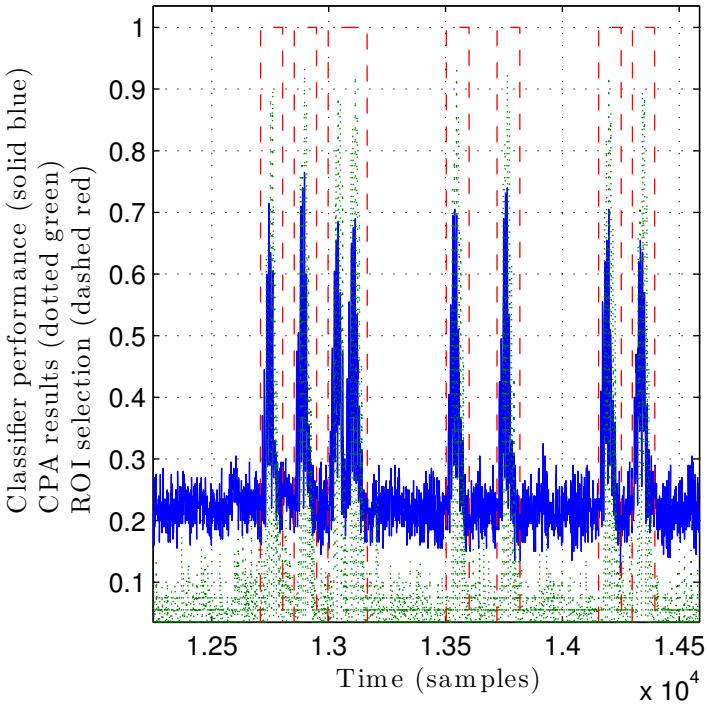

Figure 1: Three-featured classifier feature selection for a typical leak. Regions of interest are indicated by the dotted line.

used for testing exactly once. We then measured the success rate of this classifier. For each classifier i with a success rate of 50% or above we added the samples used as its input features – i, $i+8$, $i+16$ – into the set of candidate features.

2.2.2 Improving Candidate Search Efficiency

On our system, the aforementioned candidate selection process took 15 minutes per leak. Since this process needs to be repeated for each of the 84 leaks, we were interested in ways of improving its efficiency. Our general approach was to find regions of interest (ROI) in the traces, then search for features only in those regions. For this purpose we ran, for every leak, a correlation test as performed in standard CPA [8]: Every leak has a vector of size 200 of simulated intermediate leak-values. We map these values to their Hamming weight values, then compute the absolute value of the Pearson correlation coefficient between the Hamming weights and the trace vectors at each trace-step. For most locations the correlation is less than 0.15, with several peaks which are above 0.8. We assume that 8 trace-steps correspond to a measurement of a full clock cycle. Thus, if we define a trace-step neighborhood to be 3 clock cycles from each side, we need to

take into account 24 trace-steps from each side of a single trace-step. Therefore, for all trace-steps in which the correlation is above 0.4 we say that these trace-steps and their 24 neighbors, from each side, are regions of interest (ROI) for this specific leak. We have noted experimentally that good candidate features are always inside the region of interest, as illustrated in Figure 1. Therefore, we perform the candidate features search described in the Section 2.2.1 only in regions of interest. Since the regions of interest are typically less than 2% of the entire trace, this dramatically speeds up the search process – from 15 minutes down to approximately 20 seconds per leak.

2.2.3 Selecting the Best Features

At this stage we have a list of candidate features for every leak (of the 84). The number of candidate features per leak ranges from 60 to 550. We would like to find the optimal combination of these features which gives the best classification results. For this task we performed the following greedy algorithm: Assume that there are n candidate features for a specific leak. We define an n-size Boolean vector $used\_features$, s.t. $used\_features(c) = True$ if the candidate c should be used in the optimal classifier. We start our process with $used\_features$ set to $False$ for all features. Then sequentially, for every c, we set $used\_features(c) \leftarrow True$ and measure classification results using *cross-validation* of 10 groups: Divide all 200 traces into 10 groups of 20 traces each; train on $9 \cdot 20 = 180$ traces and test on 20 traces; repeat 10 times so each group is used for testing exactly once. If the classification results were better with candidate c then we leave $used\_features(c) = True$ and move on to candidate $c + 1$. Otherwise, we set $used\_features(c) \leftarrow False$ and move on to candidate $c + 1$. At the end of the process we are left with $used\_features(c) = True$ for all candidates which together marginally improve the classification results. On average, each leak was left with about 55%-65% of the candidate features. i.e., 35%-45% of the candidates are discarded in this process. The last trained classifier on a group of 180 traces is saved as the best classifier for the leak.

2.2.4 Improving Decoding Performance

The classifier we produced in the previous subsection was trained on only 5 classes, covering Hamming weights 2 through 6. However, we need to extend the classifier to classes 0 through 8. In addition, the very small amount of training data provided to

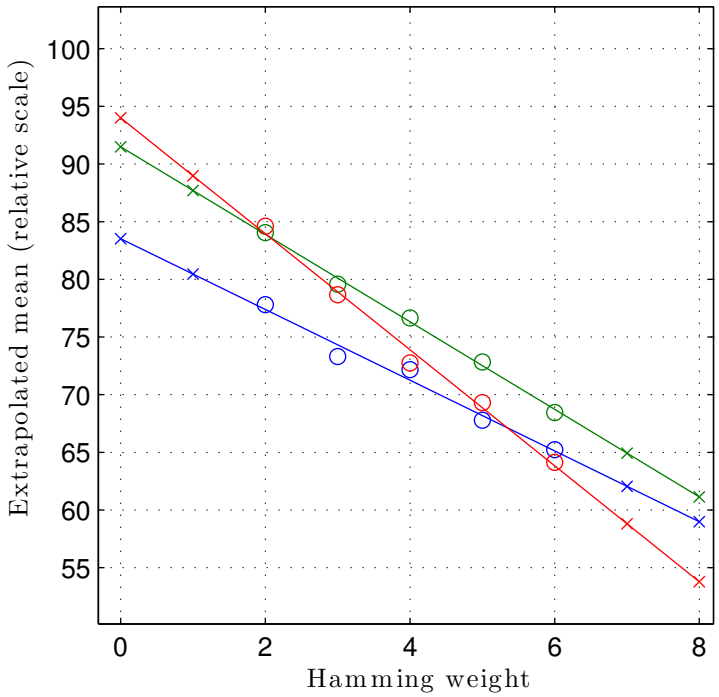

Figure 2: An example of linear regression of the mean values of 3 features. Circles indicate means output by the Naive Bayes classifier. Crosses indicate means recomputed via linear regression for classes 0 through 8.

the classifier resulted in high variation in the estimated noise σ between classes. To extend and improve the performance of our decoders, we used some of our physical knowledge of the DUT. Specifically, we made the additional assumptions that the noise is largely invariant between classes, and that the means of a certain feature among different classes are linearly related to the Hamming weight of the class.

Our naive Bayes classifier returns, for each feature, a mean and variance for each class of the 5 classes (2-6). As illustrated in Figure 2, inspecting a single feature in a classifier for a particular leak shows that the means for the 5 classes seem to form a straight line. Thus we extended our classifier: For each feature, we found a linear function that best explains the 5 means. We did this using Matlab's polyfit function for linear fitting. Using this reconstructed linear function we extrapolated from the mean values for classes 2 through 6 (marked in the figure as circles) to the mean values for classes 0,1,7 and 8 (marked in the figure as crosses). In addition, we assumed that the variance should be constant for all classes. Thus, we used polyfit to find the constant value among the 5 variances and used this value for all 9 classes (0 through 8). Using those new mean values and variances we reconstructed a new classifier (denoted the

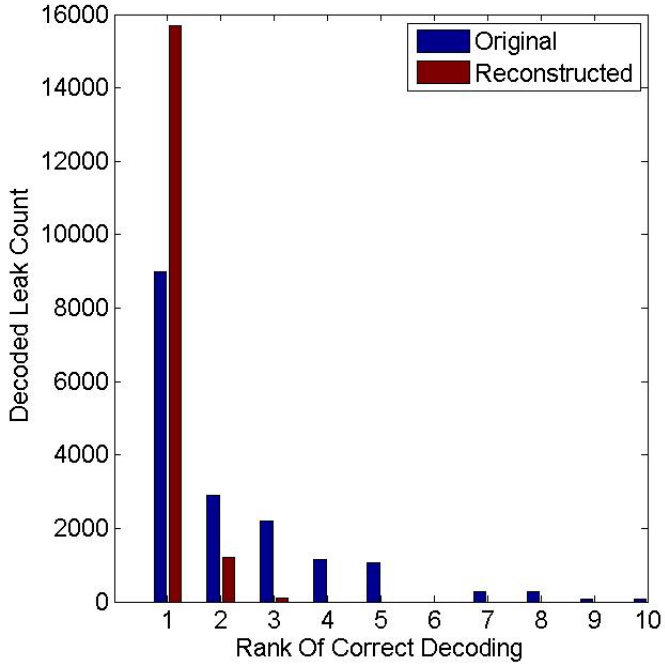

Figure 3: Ranks of correct leak values given by the classifiers. In blue are the ranks given by the 5-classes original classifiers. In solid red are the ranks given by the 9-classes reconstructed classifiers.

reconstructed classifier). The classification results seemed to improve dramatically, as described in the following subsection.

2.3 Decoding Phase

After profiling the a device that is physically identical to the attacked one, we now have classifiers which represent all the templates for the device. What is left now is to measure the attacked device and use the well-trained classifiers to estimate what the leaks are. We have 84 classifiers, each of which gives us for each trace an a-posteriori estimation of the leaked states. Thus, for each trace the decoder outputs a matrix of 84×9 probabilities.

Before trying to formulate and solve a Template-TASCA equation set, we first want to measure the quality of the decoder. For every leak, our decoder gives us 9 a-posteriori probabilities of what the Hamming weight of the state should be. We can sort these values from the most probable Hamming weight to the least probable. We say that the Hamming weight with highest probability is of rank 1, and the Hamming weight which is least probable is of rank 9. Knowing the correct leaks, we can check the ranks of

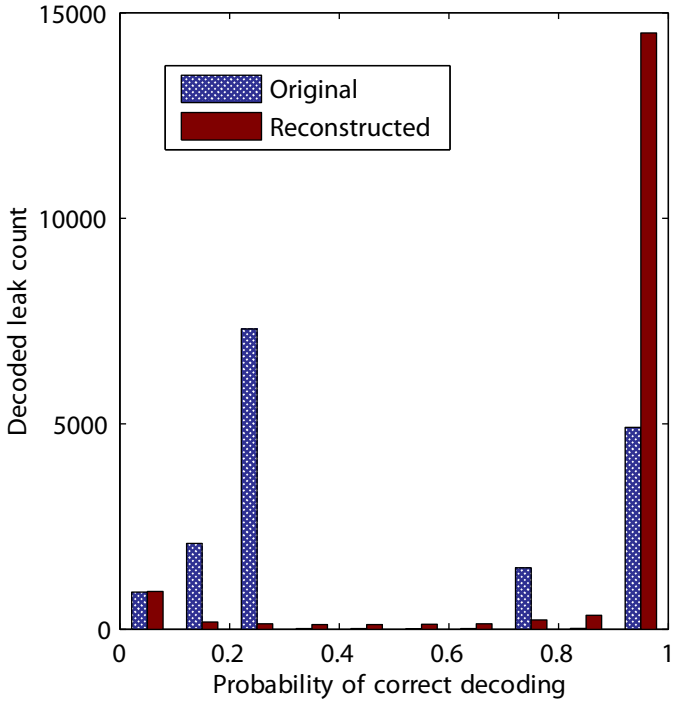

Figure 4: Probability of correct decoding by the classifiers. In hatched blue is the performance of the 5-classes original classifiers. In solid red is the performance of the 9-classes reconstructed classifiers.

the correct leaks. Ideally we would like the correct leak value to be ranked 1, i.e., to be the value with the highest possible probability. We have 84 leaks, and 200 traces which gives us $84 \times 200 = 16800$ a-posteriori estimations for the leaks. In Figure 3 we can see that in general most leaks were identified correctly among the first 3 ranks. We can also see that the reconstructed classifiers significantly pushed all the correct values to be ranked among the highest 3 probabilities. Another way to measure the performance of our decoder is to look at the probabilities given to the correct leak states. Ideally, we would like all correct leak values to have probability=1. In Figure 4 we can see that in the case of the original classifiers - only a little more than third of the leaks are estimated with high probability (probability > 0.9). The reconstructed classifiers (with regressed mean values) dramatically improved the given estimations for the correct states.

3 Reconciling the Leaks of Information - The Algebraic Way

Given a single trace, the output of the template decoder's online stage is a matrix consisting of 84 rows, each containing 9 a-posteriori probabilities for the Hamming weights of each byte of the state. In a standard template attack, the next step would simply be to select the most likely value for each position in the key and work down the list until the correct key was found [31]. However, as stated in Section 1, not all combinations of state bytes correspond to a transcript of a valid encryption. Instead, we employ a constraint solver to find the valid states which maximizes the a-posteriori probability for all state bytes simultaneously.

The solver we chose to use is SCIPspx version 1.2.0 [3–5]. SCIPspx won the first prize for non-linear optimizer in the Pseudo-Boolean Evaluation Contest of SAT 2009 [17]. SCIPspx solves the optimization problem by using integer programming and constraint programming methods. It performs a branch-and-bound algorithm to decompose the problem into sub-problems, solving a linear relaxation on each sub-problem and finally combining the results. The linear relaxation component of SCIPspx is the standalone LP solver SoPlex [33]. The solver was run on a quad-core Intel Core i7 950 at 3.06GHz with 8MB cache, running Windows 7 64-bit Edition. To take advantage of the multiple hyper-threading cores of the server, six instances of the solver were run in parallel.

We note that even though the template decoder we created could only output the Hamming weights of the internal states, the output of the algebraic stage is the state vector which contains **the complete AES key**, and not just its Hamming weight.

3.1 An equation set for AES

The AES instances we submitted to the solver were created according to the principles described in [22], and are similar in form to the example found in the appendix of [22]. In addition to the probability distribution of the leak values, the AES equation set as described in [22] expects to receive a-posteriori probabilities for the Hamming weights of the key bytes themselves as part of the state. However, our training set consisted of 200 traces created with the same key, making it impossible to create template decoders for the actual key bytes. Instead, we used the apriori distribution for Hamming weights

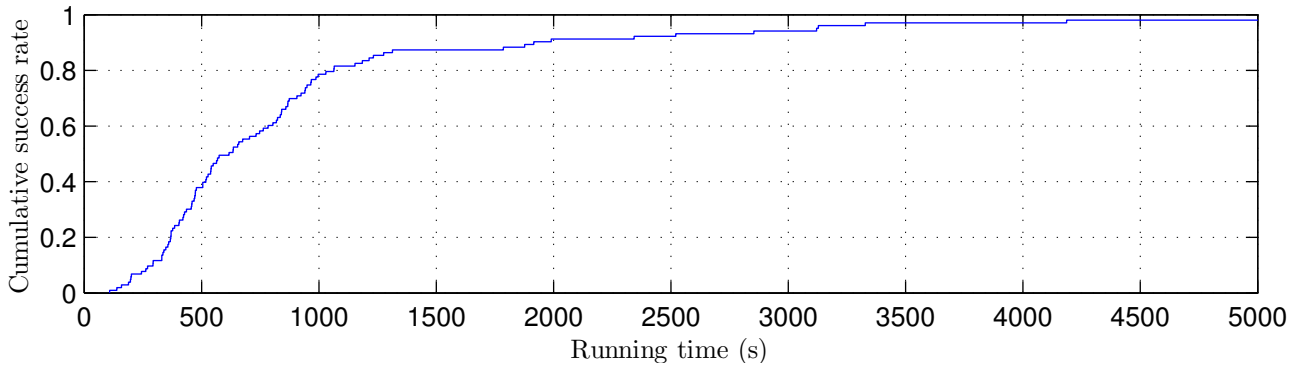

Figure 5: Running time distribution for the two-trace scenario

of an 8-bit value for the key bytes:

$$\Pr\left(HW = x\right) = \frac{\binom{8}{x}}{256} \tag{1}$$

Our experiments were performed on a pair of traces (each with a different plaintext). To create an equation set for two traces, we created two separate equation sets, then concatenated them with an additional clause which requires the keys to be identical in both states.

3.2 Combinatorial Exclusion

Since the decoder used in the template phase estimates each potential value by a Gaussian random variable, the a-posteriori probability assigned to any candidate value, no matter how unlikely, can never be exactly zero. In initial testing we discovered that while the highly unlikely values provided to the solver did not cause incorrect answers, they had the drawback of raising the running time of the solver by one or two orders of magnitude. To improve the performance of our attack, we made the engineering decision to set to zero the a-posteriori probabilities of all values which are far less likely than the least likely correct value. In our data set the lowest a-posteriori probability a correct value had was $3 \cdot 10^{-40}$. Choosing a reasonable margin of 3 orders of magnitude, we set to zero all entries whose probability was at most 10^{-43}. This reduced the amount of nonzero entries in the decoder output to 54% of the entire a-posteriori probability matrix.

4 Algebraic Reconciliation - Results

4.1 Conditions for success

As stated in [22], there are three conditions which must all hold for a Template-TASCA attack to succeed. First, the correct state must be inside the solver's solution space, that is, all bytes of the state must be assigned a nonzero a-posteriori probability by the decoder and by the combinatorical exclusion steps. Next, the solver must terminate in a reasonable time. Finally, the solver's output must be within brute-force distance (at most 4 incorrect key bytes) of the correct key.

4.2 Double-trace attack

Our first attack was performed on pairs of traces, which were combined as described in Section 3.1. Each of the 200 traces in the sample set was matched to another trace, resulting in 100 experiments. We were pleased to find that in 100% of the cases the attack succeeded in finding the correct key from two traces. In the two-trace attack the key was recovered precisely, without the need for an additional brute-forcing step. Figure 5 shows the cumulative distribution function of the running times for the two-trace attack. The median time for a successful attack was 607 seconds, while the maximum time was approximately 6 hours.

4.3 Comparison with Solver-Based Attacks

As stated in [22], there are two main methods of performing template-based algebraic attacks: Template-TASCA, which uses an optimizer, and Template-Set-ASCA, which uses a generic SAT solver. Optimizers are less efficient than solvers in terms of running time. On the other hand, a solver does not have any efficient way of representing the objective function which contains the a-posteriori probabilities. Instead, the solver-based approach restricts the set of possible solutions by using some threshold, then searches for a satisfying solution within this threshold. As shown in [22], with good-quality inputs to the algebraic phase the Set-ASCA method is much faster than the TASCA method. As the quality of the outputs from the template phase falls, the running time of the Set-ASCA method gradually becomes worse than that of the TASCA method. Finally, beyond some threshold, the Set-ASCA method is unable to recover the correct key.

We evaluated the performance of the Template-Set-ASCA method on a subset of the data. The success rate was approximately 30% for the solver, when compared to 100% for the optimizer. The running time of the solver-based approach was worse than that of the optimizer-based approach by a factor of between 2 and 10.

5 DPA contest V4

After evaluating our decoding on the WS2 data set (described in 2.2), we wanted to try our method on another scenario. We chose the DPA contest v4 [2] since it is micro-controller based and provided a large public data set. The cipher presented in the contest is an implementation of AES-256 variant, which 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.

In the traces of DPA v4 the Hamming weights of pm_i are also leaked, adding 16 new leaks to the previous 84 identified in Section 2.1, for a grand total of 100 leaked values.

The Hardware: The above cipher was implemented on an Atmel ATMega-163 smart card. The power consumption of the smart-card was sampled using a Lecroy Waverunner 6100A scope at the rate of 500MS/s. More details can be found in [2]. The DPA v4 contest published 100,000 power measurements traces of the smart card, running the

RSM-AES implementation. All traces were with the same 256-bit key, but with different plaintext being used as input on every measurement. Each power trace consists of 435,002 samples. The goal of the contest is to find the first 128 bits of the key, using the smallest number of power traces.

5.1 Profiling and Decoding

The decoding we performed on the DPA v4 data set was evolved from the decoding techniques we previously used. Three main improvements were made: The first is a new scoring technique, used to evaluate the amount of information each candidate holds on a specific leak. The second improvement was to perform a more thorough and robust search of candidate samples, to be used for classification of the leak value. Lastly, in the heuristic feature selection phase, we used a different score method to evaluate the overall performance of the multi-feature classier. Our profiling of the DPA v4 data consists of the following steps:

1. Find regions of interest: For every leak among the 84 described in Section 2.1 we want to find *regions* in the trace which contain significant information about the specific leak. These regions contain multiple trace-samples. Each sample is a possible input feature for template classifier.

2. Calculate features' scores: For every candidate feature (i.e., trace sample) found in previous step we want to give a *score* representing how much information does this single feature convey on the specific leak, independent of any other features.

3. Find best combination of features: From the features with the highest scores - select a combination which can be used together to classify the value of a specific leak.

4. Train classifiers for Hamming weights 2-6: Train template classifiers for every leak. The input for the classfiers is the features' combination selected in the previous step.

5. Extend the classifiers to be able to classify Hamming weights 0-8, as described in Section 2.2.4.

6. Measure the performance of the template decoder of every leak.

We now elaborate on the specifics of each step.

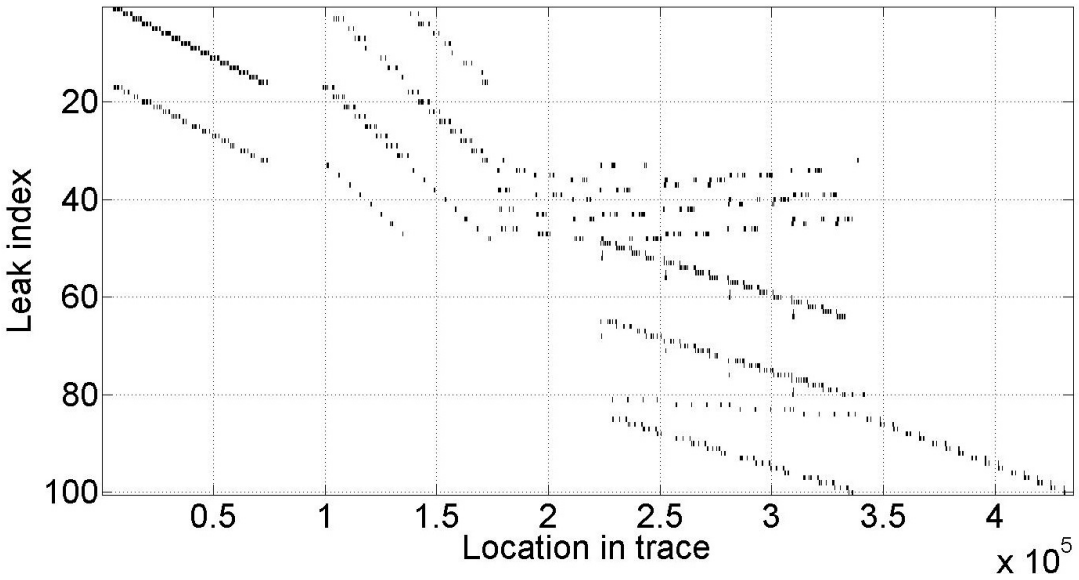

Figure 6: Regions of interest per leak, evaluated over 200 traces. Leaks 0-15 are for Hamming-weights of pm_i, 16-31 are for AddKey outputs, 32-47 are for SubBytes, 48-63 are for $x2$, 64-79 are for xt, 80-83 are for $x4$ and 84-99 are for MixColumns outputs.

5.2 Finding Regions of Interest

The process was performed is similar to the process described in Section 2.2.2. As in our previous experiment, the correlation values on the Hamming weights of the leaks showed significant peaks. Therefore, we chose the power leak model to be the Hamming weight model. The correlation threshold that was used to spot regions of interest was 0.45 (remember that the absolute correlation value is between 0 and 1). For every location in the trace with correlation above 0.45 we took all trace-locations in neighborhood of 200 samples (compared to a neighborhood of 24 samples in Section 2.2.2), from each side, to be the *region of interest* - ROI. The results are depicted in Figure 6. Each single horizontal line represents the ROIs in a trace for a specific leak, similar to the ROI in Figure 1. We can see that the ROIs for the values of pm_i (leaks 0-15) occur earlier in the trace than the ROIs of $x2$ and xt values (leaks 48-79), which occur earlier than the ROIs of MixColumns output bytes (leaks 84-99).

5.3 Calculate Features' Scores

The purpose of this step is to give each individual sample in the trace a score, representing the amount of information it contains on a specific leak. We only calculate the score for features within the *regions of interest* we found in Section 5.2, of the specific

leak we evaluate. We chose to calculate to score in the following manner:

1. Train a Bayesian classifier for Hamming weights 2-6 based on a single feature. The input feature is the sample in the trace we currently evaluate. The training is done on 200 traces.

2. Expand the Bayesian classifier to distinguish between Hamming weights 0-8 as explaind in Section 2.2.4.

3. Evaluate the classifier performance on 200 other traces . Notice that a classification run on a trace yields a 9-probabilities vector representing the likelihood of each class of the 9 possible Hamming weights.

4. For every trace among the 200 in step 3 we have the correct leak value, i.e., the correct Hamming weight of the intermediate computation. For the correct Hamming weight we can look at the a-posteriori probability the classifier gave it in step 3. In total, we have a vector of 200 a-posteriori probabilities. An ideal classifier would produce a vector containing only perfect probabilities - 200 values of 1. That is, it assigns the correct Hamming weight an a-posteriori probability of 1, and the 8 incorrect Hamming weights would receive an a-posteriori probability of zero. The feature score is set to be the average of the 200 a-posteriori probabilities, assigned to the correct Hamming weight by the classifier trained on this single feature. Figure 7 depicts the features' score on the first byte after *AddRoundKey*.

There are several reasons for selecting the a-posteriori probability as the feature score. First, if the classifier selects the correct Hamming weight - that means the a-posteriori probability is higher than the probability of the other Hamming Weights. Secondly, even if the classifier did not select the correct Hamming weight as the most probable, the fact that the classifier assigns a high a-posteriori probability to it means the feature (sample) indeed contains information about the leak.

Figure 8 depicts 100 histograms, 1 for each leak. Each vertical line represents a histogram similar to those in Figure 3. As seen in the color bar to the right, the brightness represents the amount of traces in each bin, ranging from 0 to 600. It can be seen that for all 100 leaks (vertical lines), for most of the 600 traces the correct

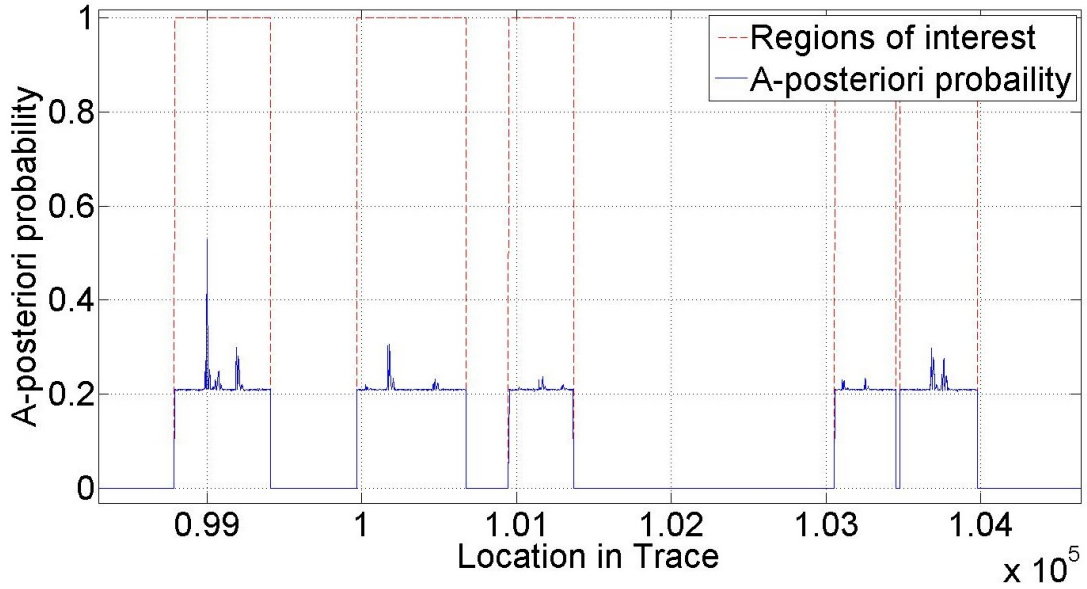

Figure 7: Features' scores for leak of first byte after *AddRoundKey* (within the regions of interest)

Hamming weight values were classified among the top 3 high a-posteriori probability classes (thus the top 3 horizontal lines are very dark).

5.4 Combining the Best Features

The purpose of this step is to select a subset of the features (trace samples) to be used as input for the classifier. In the previous step we assign each feature its own score. We now try to find a group of features that together holds the most information regarding the specific leak. For this we use a process similar to Section 2.2.3. For each leak, for all candidate input features (trace samples) in *regions of interest* we have a score. We used the following greedy heuristic:

1. Sort all features by score, and keep only the top 500 features to save running time.
2. Set $used\_features \leftarrow \{\}$, and $last\_grade \leftarrow 0$
3. Going in descending score order, for every feature $f_i \in sorted\_features$
   (a) $used\_features \leftarrow used\_features \cup f_i$
   (b) $group\_grade \leftarrow Grade(used\_features)$
   (c) • If $group\_grade > last\_grade$: $last\_grade \leftarrow group\_grade$

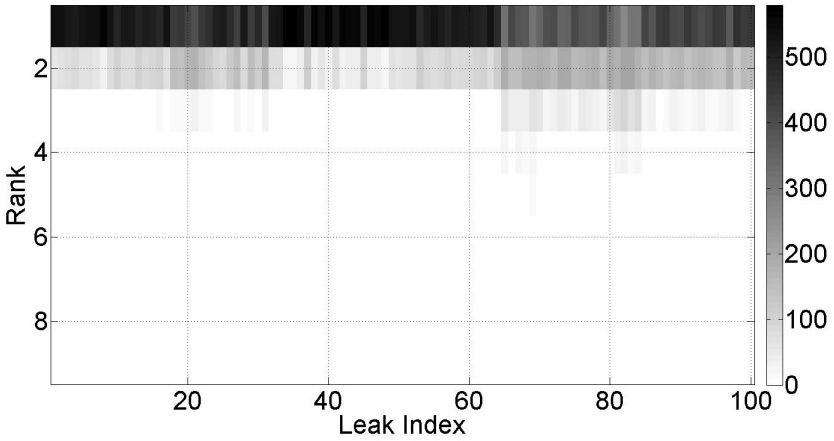

Figure 8: Classification ranks of the correct Hamming weight over 600 traces. Each single vertical line is a histogram of the distribution of ranks for a specific leak, similar to the histograms in Figure 3. Leaks 0-15 are for Hamming-weights of pm_i, 16-31 are for AddKey outputs, 32-47 are for SubBytes, 48-63 are for $x2$, 64-79 are for xt, 80-83 are for $x4$ and 84-99 are for MixColumns outputs.

- else: $used\_features \leftarrow used\_features \setminus f_i$

$Grade(used\_features)$ is calculated similarly to step 4 in 5.3: We train a Bayseian classifier whose input features are those in $used\_features$, trained on 200 traces. We then calculate the a-posteriori probability of the correct Hamming weight over 200 traces, different from those used for training. As before, the grade is the average a-posteriori probability. At the end of the process $used\_features$ represents the best found combination of features, that shall be used for the classification of the specific leak. Figure 9 is a zoom in on Figure 7, with green circles marking the selected features. Figure 10 shows how many features were selected for each classifier.

5.5 Train, Extend & Measure performance

In the training step we train a Bayesian classifier for each leak, using the selected features from Section 5.4. The classifiers are trained to distinguish between Hamming weights 2-6. In the extending step, we perform the process described in Section 2.2.4 to extend the classifiers such that they will be able to distinguish between Hamming weights 0-8. The training process was performed on 200 traces. We denote these traces *training traces*. In Section 5.3 we used another 200 traces to evaluate a *score* for each feature. We denote these traces *evaluation traces*. In Section 5.4 we used the **same** 200 *evaluation traces* in order to evaluated a mutual score of all features in $used\_features$.

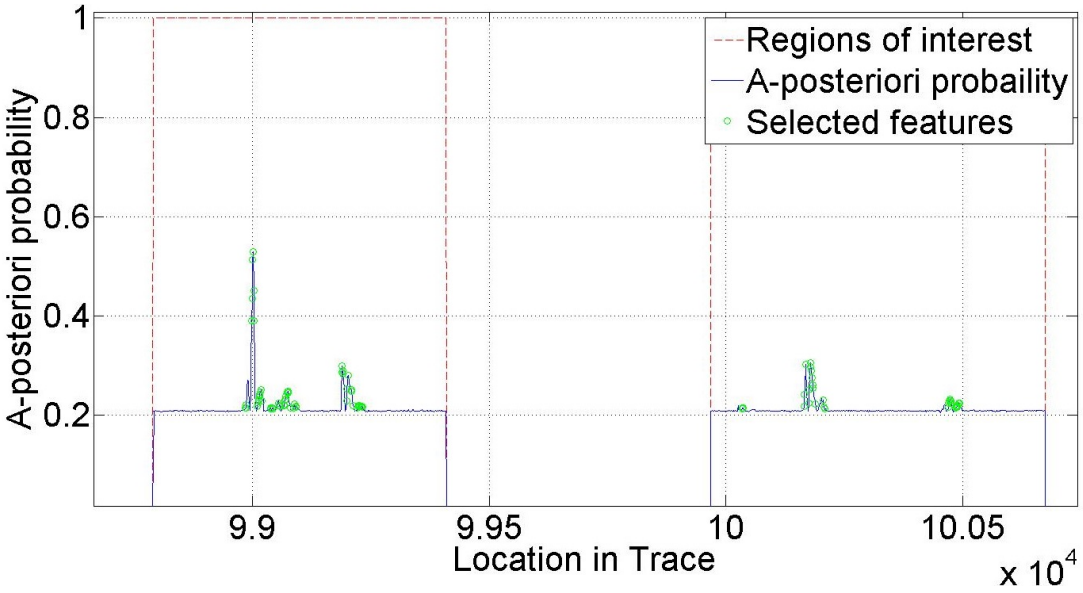

Figure 9: Features' scores for leak of first byte after *AddRoundKey*. The circles are marking the features which were selected to be used for classifying this leak.

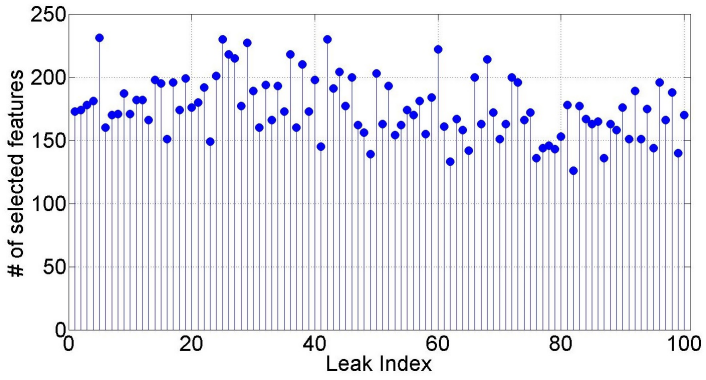

Figure 10: Number of features heuristically selected for each leak

Until now we used 400 traces in total. In order to test the performance of our "final" classifiers we used another 600 traces, denoted *performance test traces*. These traces are different from the previous 400, in order to avoid over optimistic performance evaluation.

As we recall from Section 2.1, there are 84 leaks of information, and the RSM counter measure adds additional 16 interesting leaks (see description of pm_i values at the beginning of Section 5). Figure 11 depicts the a-posteriori probabilities of the 100 correct Hamming weight values of 100 leaks, as assigned by the classifiers, over 600 traces. We can see that the classifiers of leaks 1-64 and 84-99 classified the correct leak value with a-posteriori probability close to 1, for more than 50% of the traces. For

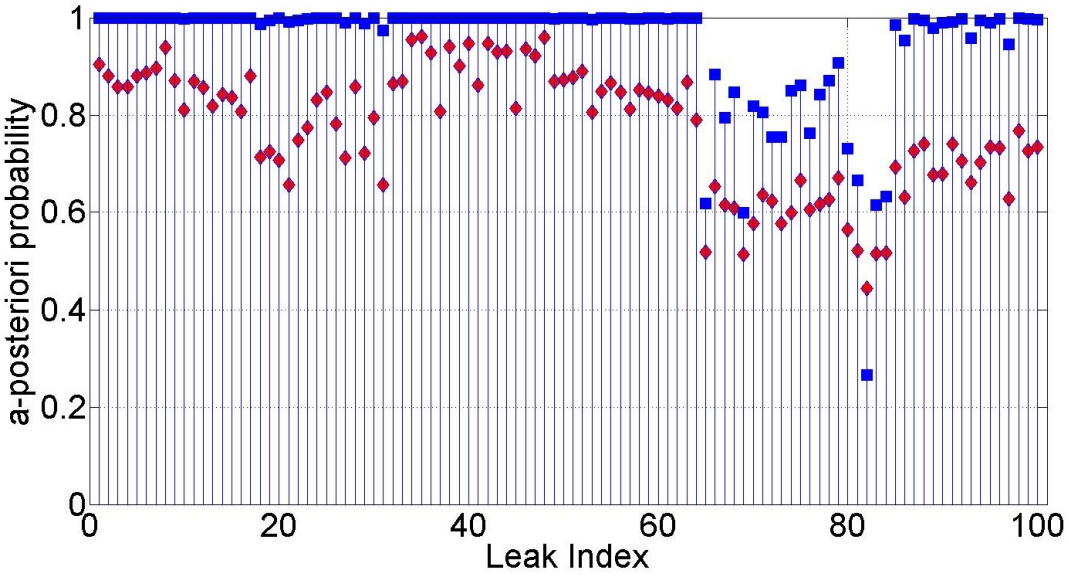

Figure 11: Average and median values of a-posteriori probabilities of the correct leak values from 100 classifiers, over 600 traces. Diamonds represent the average values; squares represent the median values. Leaks 0-15 are for Hamming-weights of pm_i, 16-31 are for AddKey outputs, 32-47 are for SubBytes, 48-63 are for $x2$, 64-79 are for xt, 80-83 are for $x4$ and 84-99 are for MixColumns outputs

all leaks, the average probability of the correct values is significantly more than 10^{-1}. This will be significant in the following solving phase.

6 New Approach for Reconciling Side Channel Information

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. [20] 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 [22] 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.

6.1 Reconciling Side Channel Leaks via a Constraint Solver

Instead of using a general SAT solver we developed a specialized constraint solver. The constraint solver embeds a model of the encryption process, accepts the known plaintext, and the output of the *decoder*, and outputs the highest probability keys with an estimation of their likelihood. However, unlike the algebraic attacks of [28] and [22], our constraint solver is not a general purpose Pseudo-Boolean or SAT-solver.

We wrote a special solver that is focused 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 [22], it is a probability distribution over the possible values of X. The decoder is usually constructed as a template decoder [9]. As in [22] 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 [34] and [22]).

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.

7 Probabilistic Methodology

7.1 Dissecting an Encryption Algorithm

An encryption algorithms is a function $f(p,k) = c$, where p is the plaintext, k is the secret key and c is the cipher text. Our work is in a "known plaintext" scenario, thus p and c are known. The computation of f consists of several subcomputations, i.e., f can be viewed as $f = f_1 \circ f_2 \circ ... \circ f_n$. In a byte-oriented cipher these functions f_i can generally be divided into one of two classes: single-input functions $f(X) \to Z$ or dual-input functions $f(X,Y) \to Z$ where X, Y and Z are all 8-bit quantities. In a side channel attack model we assume to have some partial knowledge on the distribution of the intermediate values X, Y, Z. Having some non-trivial information of the probability of possible values of Z can be used to infer some knowledge on X, Y and vice versa. Our constraint framework provides a simple set of tools that enables reconciling all the partial knowledge acquired by the side channel leakage into a single conclusion (distribution) on the possible values of k.

7.2 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 [10, 12–14]. In particular, Hill [12] 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 $\hat{P} = \&(P_{E_1}, P_{E_1}) = (\hat{p}_1, .., \hat{p}_N)$ 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 [12] 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.

7.3 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:

$$\hat{P}(X = a) = \frac{1}{\gamma} P_{E_1}(X = a) \cdot P_{E_2}(f(X) = f(a)) \tag{2}$$

7.4 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:

$$\begin{aligned}
&\hat{P}(X = a, Y = b) = \\
&\hat{P}(c) \cdot \frac{P_X(a) \cdot P_Y(b)}{\sum_{k,l:f(k,l)=c} P_X(k) \cdot P_Y(l)} = \\
&\hat{P}(c) \cdot \frac{P_X(a) \cdot P_Y(b)}{P_f(c)} = \\
&\frac{1}{\gamma} P_f(c) P_Z(c) \cdot \frac{P_X(a) \cdot P_Y(b)}{P_f(c)} = \\
&\frac{1}{\gamma} P_X(a) P_Y(b) P_Z(c)
\end{aligned} \tag{3}$$

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

$$\hat{P}(X = a, Y = b) = \frac{1}{\gamma} P_X(a) P_Y(b) P_Z(f(a, b)) \tag{4}$$

8 Building Blocks

8.1 Capturing the Information Flow

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 cipher 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.

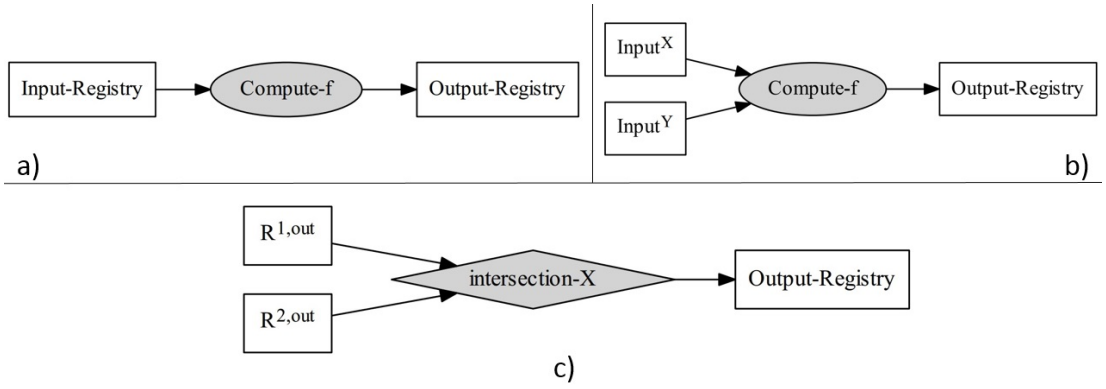

Figure 12: Illustration of three types of constraints: a) single-input constraint, b) dual-input constraint, c) data-redundancy constraint

8.2 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.

Registry Nodes: 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 $P(f(X) = v_n) = \alpha^{out}_{v_n}$.

Compute Node: We connect the input registry to the output registry via the *Compute-f* node (see Figure 12a), 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 7.3:

$$\hat{\alpha}_n = \tfrac{1}{\gamma} P(X = b_n) \cdot P(f(X) = f(b_n)) = \tfrac{1}{\gamma} \cdot \alpha^{in}_{b_n} \cdot \alpha^{out}_{f(b_n)}$$

for a normalization factor γ. After the computation is done the *Output-Registry* contains tuples of the form $(b_n, f(b_n), \hat{\alpha}_n)$.

8.3 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 12b). 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 7.4, 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.

8.4 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.

8.5 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 12c). 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.

8.6 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. The beginning of the flow is the first unknown values - the key bytes. Each key byte requires a *registry* containing all possible candidates for that key byte. The probability distribution is either derived from a side channel leak, or it is uniform if nothing is known. 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 8.2. Note that the outputs of a registry may be used as inputs for more than one computation. We continue to chain single-input constraints until we reach a dual-input computation. We then use the dual-input constraint (Section 8.3) 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.

9 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 the same 84 values as described in 2.1. For each leaked byte our decoder (see Section 10.1) produces a probability distribution over the 256 possible values.

9.1 Limited Diffusion

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.

9.2 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 computations. 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 13 illustrates the single-input constraints for four key bytes.

9.3 Basic Computation of MixColumns

A common implementation of the MixColumns computation in software on an 8-bit microcontroller (cf. [29]) is to compute the following intermediate values:

1. The XOR value of four column bytes:

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

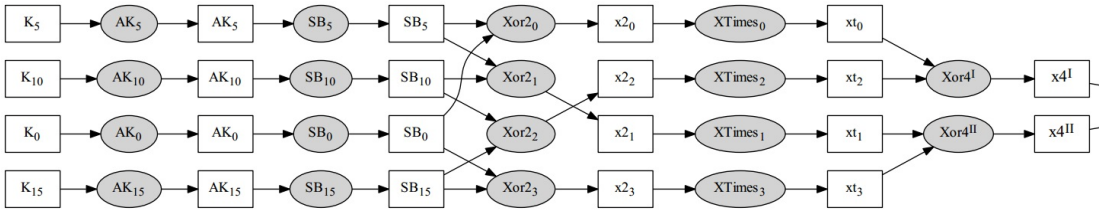

Figure 13: Visual representation of the constraint solver tracking four key bytes up to the X4 computation in AES. Registry nodes are drawn as rectangles and compute nodes as ellipses. Abbreviations: AK-AddKey, SB-SubBytes

2. The XOR values of adjacent bytes:

$$x2_0 \leftarrow b_0 \oplus b_1$$
$$x2_1 \leftarrow b_1 \oplus b_2$$
$$x2_2 \leftarrow b_2 \oplus b_3$$
$$x2_3 \leftarrow b_3 \oplus b_0$$

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 13).

9.4 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

9.5 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. For option I we get: xt_0-Registry→Compute-$x4^I$, xt_2-Registry→Compute-$x4^I$. For option II we get xt_1-Registry→Compute-$x4^{II}$, xt_3-Registry→Compute-$x4^{II}$. Figure 13 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 8.5) 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

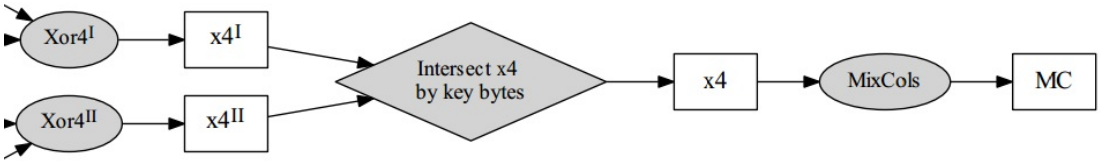

Figure 14: Visual representation of the constraint solver tracking four key bytes, of column 0, from $x4$ to MixColumns computation. MC stands for MixColumns

normalized so that all probabilities sum to 1. The final result is the MC registry. Figure 14 illustrates the constraint solver from $x4^I$, $x4^{II}$ registries to the MC registry.

9.6 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 [32], 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.

10 Performance Evaluation

10.1 Decoding

To profile the power consumption behavior of the RSM-AES implementation we used the techniques described in Section 5. Our leak model is the Hamming-weight model. 100 classifiers were trained to classify 100 intermediate values. Of these, 84 intermediate values are those described in Section 2.1, and 16 values are the masked plaintext bytes of the RSM counter-measure (see description of pm_i values at the beginning of Section

5).

For each power trace, we can calculate how many classifiers (out of 100) incorrectly predicted the Hamming-weight. We evaluated this quantity over 600 power traces. For more than 90 % of the traces no more than 30 out of 100 classifiers are wrong, thus allowing the constraint solver to overcome the classifications errors. Note further, 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, i.e., it has a high rank (see Figure 8), it helps the solver distinguish the correct values from the incorrect ones. As we shall see, even with far-from-perfect classifiers, our framework is able to find the correct keys.

10.1.1 Overcoming the RSM Counter Measure:

As described in Section 5, 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. For a given Hamming weight 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.

10.1.2 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 $S_{hw} = \|\{x \in \{0..255\} | HW(x) = hw\}\|$ 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 $\sum_{x=0}^{255} P(b_l = x) = 1$.

10.2 Implementation of the Constraint Solver

The custom solver designed for AES as described in Sections 8 and 9 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 10.1.2. These are the α^{out} values described in Section 8.2.

10.3 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 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 [32] 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 [32] on our results. Instead, we opted to use a second power trace and intersect the candidate key-quartets (see below).

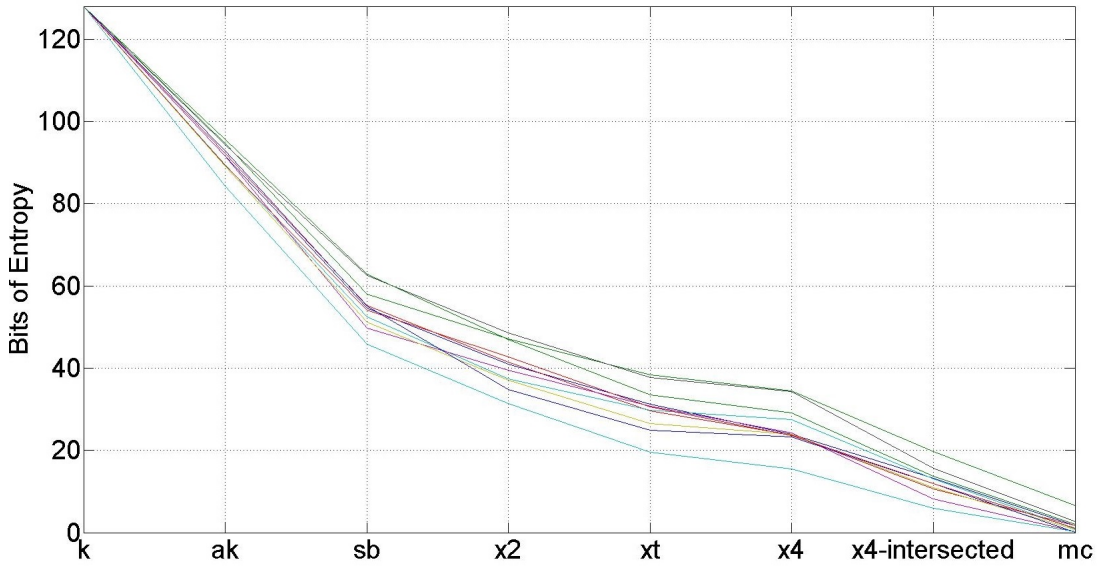

Figure 15: Evolution of entropy of 16 key bytes at different solver phases, for 10 runs on different traces. Abbreviations: ak - AddKey, sb - SubBytes, x2 - XOR of 2 bytes, xt - xtime, x4 - XOR of 4 bytes, mc - MixColumns.

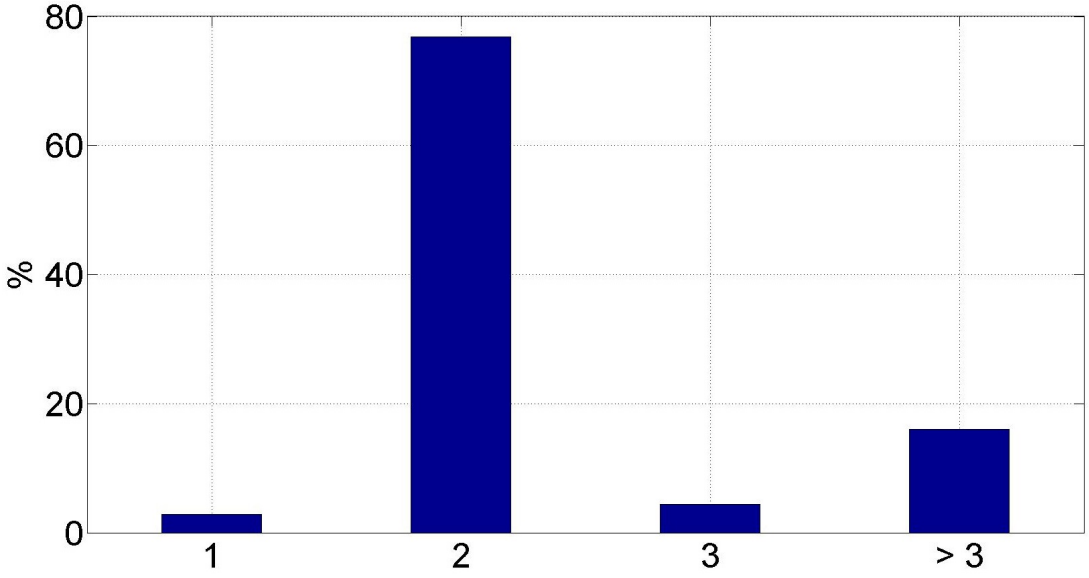

Figure 16: Number of power traces needed to find the correct key

10.3.1 Entropy:

Figure 15 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 15 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.

10.3.2 Using More Than One Power Trace:

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 16 shows how many power traces were required to yield the correct key as the first ranked 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. As of the date of writing, our solver is one of the leading entries in the contest.

11 Conclusions and Future Work

This work shows how the Template-TASCA approach first described in [22] can be put to practical use as a complement to a traditional template attack, dramatically reducing the required data complexity, both in the offline and in the online phase. The reduced data complexity means that template attacks can be applied to additional attack scenarios where access to the DUT is more restricted. It would be interesting to find ways to further reduce the data complexity of the attack.

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

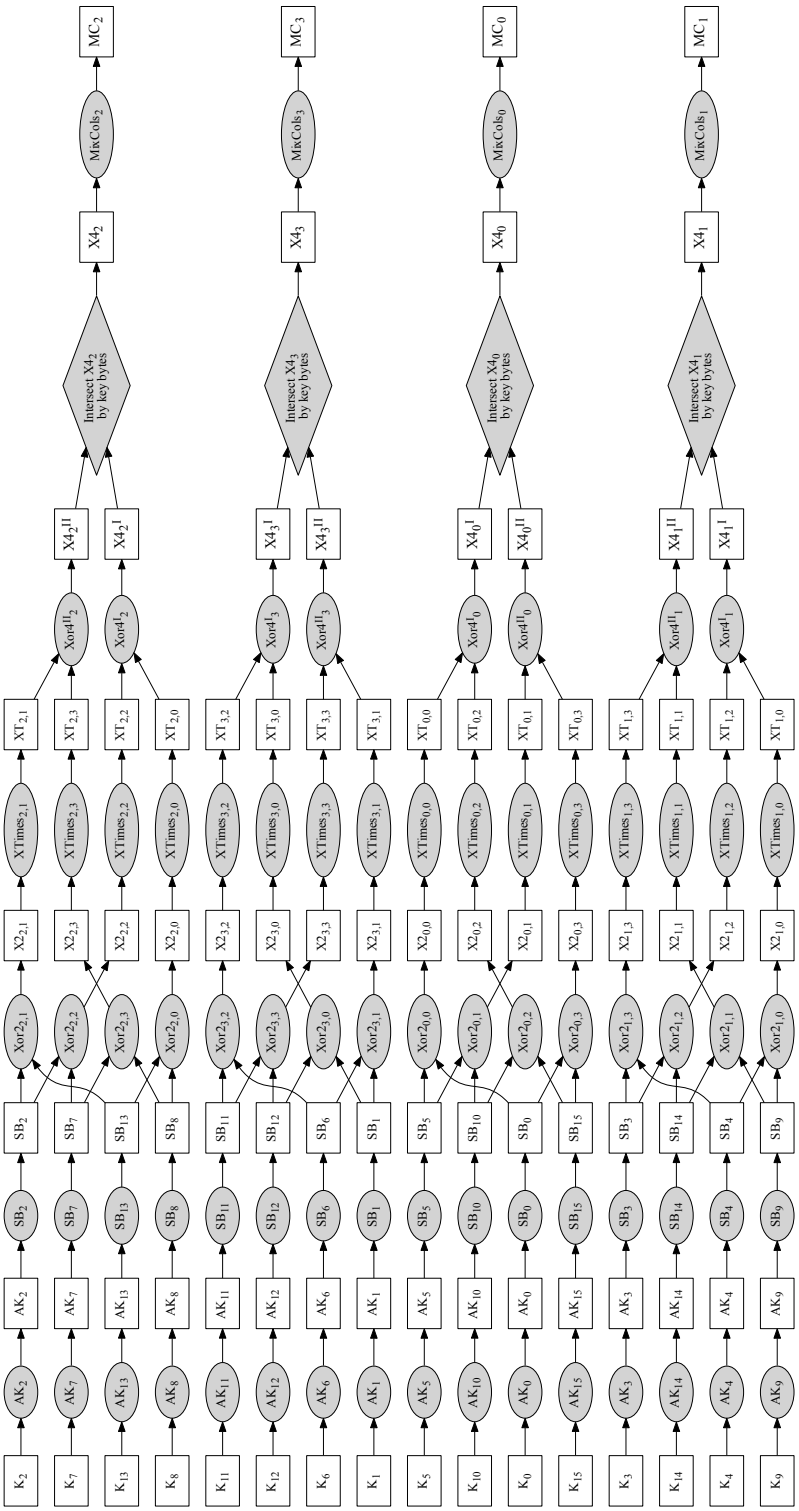

Figure 17: Flow Graph of the Full AES Constraint Solver

References

[1] Description of the masked AES of the DPA contest v4. http://www.dpacontest.org/v4/data/rsm/aes-rsm.pdf. 5

[2] DPA contest v4. http://www.dpacontest.org/v4/. 1.2.2, 5, 5, 6.1

[3] T. Achterberg. *Constraint Integer Programming*. PhD thesis, Technische Universität Berlin, 2007. 3

[4] T. Berthold, S. Heinz, and M. E. Pfetsch. Nonlinear pseudo-boolean optimization: Relaxation or propagation? In *SAT 2009*, pages 441–446, 2009.

[5] Timo Berthold, Stefan Heinz, Marc E. Pfetsch, and Michael Winkler. SCIP – Solving Constraint Integer Programs. SAT 2009 competitive events booklet, 2009. 3

[6] Andrey Bogdanov, Lars R Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew JB Robshaw, Yannick Seurin, and Charlotte Vikkelsoe. Present: An ultra-lightweight block cipher. In *Cryptographic Hardware and Embedded Systems-CHES 2007*, pages 450–466. Springer, 2007. 1.1

[7] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *Cryptographic Hardware and Embedded Systems-CHES 2004*, pages 16–29. Springer, 2004. 1

[8] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *CHES*, pages 16–29, 2004. 3, 2.2.2

[9] Suresh Chari, Josyula R Rao, and Pankaj Rohatgi. Template attacks. In *Cryptographic Hardware and Embedded Systems-CHES 2002*, pages 13–28. Springer, 2003. 1, 1.1, 1.2.2, 6.1

[10] Robert T Clemen and Robert L Winkler. Combining probability distributions from experts in risk analysis. *Risk analysis*, 19(2):187–203, 1999. 7.2

[11] M Abdelaziz Elaabid and Sylvain Guilley. Practical improvements of profiled side-channel attacks on a hardware crypto-accelerator. In *Progress in Cryptology–AFRICACRYPT 2010*, pages 243–260. Springer, 2010. 1.1

[12] Theodore Hill. Conflations of probability distributions. *Transactions of the American Mathematical Society*, 363(6):3351–3372, 2011. 7.2

[13] Geoffrey E Hinton. Training products of experts by minimizing contrastive divergence. *Neural computation*, 14(8):1771–1800, 2002.

[14] Joseph M Kahn. A generative bayesian model for aggregating experts' probabilities. In *Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pages 301–308. AUAI Press, 2004. 7.2

[15] Paul Kocher, Joshua Jaffe, and Benjamin Jun. *Differential Power Analysis*, volume 1666 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1999. 1.1

[16] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards (Advances in Information Security)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007. http://dpabook.org/onlinematerial/. 1, 1.1, 2.1, 2.1, 2.2

[17] Vasco Manquinho and Olivier Roussel. Pseudo-Boolean Competition 2009. Online, July 2009. 3

[18] Mohamed Saied Emam Mohamed, Stanislav Bulygin, Michael Zohner, Annelie Heuser, Michael Walter, and Johannes Buchmann. Improved algebraic side-channel attack on AES. *Journal of Cryptographic Engineering*, 3(3):139–156, 2013. 1.1

[19] 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. 1.1

[20] Yossef Oren. *Secure hardware - physical attacks and countermeasures*. PhD thesis, Tel-Aviv University, Isreal, 2013. https://www.iacr.org/phds/?p=detail&entry=893. 1.1, 6

[21] Yossef Oren, Mario Kirschbaum, Thomas Popp, and Avishai Wool. Algebraic side-channel analysis in the presence of errors. In *Cryptographic Hardware and Embedded Systems, CHES 2010*, pages 428–442. Springer, 2010. 1.1, 2.2

[22] Yossef Oren, Mathieu Renauld, François-Xavier Standaert, and Avishai Wool. Algebraic side-channel attacks beyond the hamming weight leakage model. In *Cryptographic Hardware and Embedded Systems–CHES 2012*, pages 140–154. Springer, 2012. (document), 3, 1.1, 1.2.2, 2.2, 3.1, 4.1, 4.3, 6, 6.1, 11

[23] Yossef Oren, Ofir Weisse, and Avishai Wool. 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*, page 7. ACM, 2013. 1.2.1

[24] Yossef Oren, Ofir Weisse, and Avishai Wool. A new framework for constraint-based probabilistic template side channel attacks. 2014. 1.2.2

[25] David Oswald and Christof Paar. Improving side-channel analysis with optimal linear transforms. In *Smart Card Research and Advanced Applications*, pages 219–233. Springer, 2013. 1.1

[26] Christian Rechberger and Elisabeth Oswald. Practical template attacks. In *WISA*, pages 440–456, 2004. 2.2.1

[27] M. Renauld and F.X. Standaert. Algebraic side-channel attacks. In *Information Security and Cryptology: 5th International Conference, Inscrypt 2009, Beijing, China, December 12-15, 2009. Revised Selected Papers*, volume 6151 of *Lecture Notes in Computer Science*, pages 393–410. Springer, 2010. 3, 1.1

[28] Mathieu Renauld, François-Xavier Standaert, and Nicolas Veyrat-Charvillon. Algebraic side-channel attacks on the AES: Why time also matters in DPA. In *Cryptographic Hardware and Embedded Systems-CHES 2009*, pages 97–111. Springer, 2009. 1, 3, 1.1, 1.2.2, 6.1

[29] W. Stallings. *Cryptography and Network Security ch. 5.* Pearson, 6th edition, 2014. 9.3

[30] Takeshi Sugawara, Naofumi Homma, Takafumi Aoki, and Akashi Satoh. Profiling attack using multivariate regression analysis. *IEICE Electronics Express*, 7(15):1139–1144, 2010. 1.1

[31] Nicolas Veyrat-Charvillon, Benoît Gérard, Mathieu Renauld, and François-Xavier Standaert. An optimal key enumeration algorithm and its application to side-channel attacks. In *Selected Areas in Cryptography*, pages 390–406, 2012. 3

[32] Nicolas Veyrat-Charvillon, Benoît Gérard, Mathieu Renauld, and François-Xavier Standaert. An optimal key enumeration algorithm and its application to side-channel attacks. In *Selected Areas in Cryptography*, pages 390–406. Springer, 2013. 9.6, 10.3

[33] R. Wunderling. *Paralleler und objektorientierter Simplex-Algorithmus*. PhD thesis, Technische Universität Berlin, 1996. 3

[34] Xinjie Zhao, Tao Wang, Shize Guo, Fan Zhang, Zhijie Shi, Huiying Liu, and Kehui Wu. SAT based error tolerant algebraic side-channel attacks. In *2011 Conference on Cryptographic Algorithms and Cryptographic Chips, CASC*, 2011. 3, 1.1, 1.2.2, 6.1