Practical Template-Algebraic Side Channel Attacks with Extremely Low Data Complexity

Yossef Oren, Ofir Weisse and Avishai Wool
Cryptography and Network Security Lab
School of Electrical Engineering
Tel-Aviv University, Ramat Aviv 69978, Israel
{yos|ofir42}@eng.tau.ac.il, yash@acm.org

ABSTRACT

Template-based Tolerant Algebraic Side Channel Attacks (Template-TASCA) were suggested in [20] 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 the first 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.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Miscellaneous

General Terms

Hardware side-channel exploits and modeling, analysis of real attacks and threat evaluation, smart-card security

1. INTRODUCTION

Side-channel attacks are attacks which reveal the secrets of cryptographic devices by observing their physical properties[14]. In this report we deal specifically with **power analysis attacks** on an **AES-128 implementation**. In this setting the secret is the 16-byte AES-128 key, while the physical property which we observe is the **power trace** of the device under test (DUT).

One popular form of side-channel attack is a type of profiling attack called the **template attack**[8, 22]. Template attacks operate in two phases – an offline phase and an online phase. In the offline phase, a reference device which is similar to the DUT but entirely under the control of the attacker is profiled and characterized, and a series of **templates** is constructed. These attacker first identifies locations in the power trace (known as interesting points) where individual sensitive elements of the device state (for example the AES key bytes) are manifest in the DUT's power consumption. Next, the attacker characterizes the particular values each such key byte can take, by matching each potential value to a multivariate Gaussian distribution over the corresponding set of interesting points. In some cases each possible value is described by its own template, while in other cases a set of similar values (such as 8-bit values sharing the a similar Hamming weight) can be grouped into the same template. The offline phase thus outputs a series of **template decoders**, each of which maps a certain set of interesting points to a certain set of secret values.

In the online phase, one or more power traces of the DUT itself are measured, and the template decoders created in the offline phase are applied to these traces. Next, signal classification techniques such as maximum-likelihood decoding are used to determine which of the secret values was the most likely to have caused these particular power traces to be emitted by the DUT.

Stated formally, the online phase of the template attack receives as input a **trace** x and outputs for each possible candidate key k_i and each possible value (or set of values) v_j the probability that the DUT would output the trace x, conditioned on the key k_i having the value v_j:

$$\Pr(x|k_i = v_j) \tag{1}$$

By evaluating this probability for all possible values of v_j and applying Bayesian inversion, the attacker can discover the value most likely to cause the device to emit this trace:

$$k_i = \arg\max_{v_j} \Pr(k_i = v_j|x) =$$

$$\arg\max_{v_j} (\Pr(x|k_i = v_j) \cdot \Pr(k_i = v_j)) \tag{2}$$

After each key byte k_i is extracted in this method, the most likely sub-keys (or some subset of less likely keys, as shown in [26]) are combined to form the complete key k.

One of the main drawbacks of template attacks is their high data complexity. Practical attacks such as [21] require millions of traces, both for the offline profiling phase and for the online attack phase[1]. It is thus interesting to search for

[1]If the attacker knows in advance that the leakage model is the Hamming weight model, fewer traces can be used in the preprocessing step; However, this modified attack does not return the complete secret key but only its vector of Hamming weights.

ways of carrying out template attacks using less traces.

1.1 Motivation

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 [7] and similar side-channel attacks. The objective of the template-algebraic method is to construct a template attack that chooses 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.

A minimal modification to the offline phase of the template attack allows the creation of templates for the entire state as well as for the key. Similarly, the online phase of a template-algebraic attack begins just like a standard template attack, by recovering the probability that the trace x was caused by each individual state byte s_i receiving a value v_j. Note that the state is dependent on both the plaintext and the key.

$$\Pr\left(x|s_i = v_j, p\right) \tag{3}$$

Using Bayesian inversion, the attacker can then transform these probabilities into a vector of aposteriori probabilities $\Pr\left(s_i = v_j|x,p\right)$. However, in contrast to standard template attacks, it is not simply possible to choose the most value for each state byte and combine them, since not all full state vectors correspond to valid encryptions. The attacker's goal is thus to search among all state vectors which correspond to **valid encryptions**, and to select among these valid states a state s^* which **maximizes the aposteriori probability** $\Pr\left(s_i = s_i^*|x,p\right)$ for all state bytes simultaneously. More formally, the objective of the online attack phase can now be stated as follows:

$$k = \arg\max_{k^*} \prod_{i=1\cdots m} \Pr\left(s_i = s_i^*|x,p\right), \text{ s.t. } s^* = State\,(k^*,p) \tag{4}$$

The added restriction on the validity of the state vector, and the fact that k does not explicitly appear in the probability calculation, means that standard maximum-likelihood methods cannot be directly applied to this problem.

1.2 Contributions

In our work we present the first practical evaluation of the template-algebraic side-channel attack, a profiling attack which can find the optimal key based on the aposteriori 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.

Document Structure The following section briefly describes the structure of the template-algebraic attack. Section 3 describes the experiment setup and leakage model. Section 4 discusses how to construct decoders for individual state bytes with a limited data set. Section 5 contains the solver-specific description of how algebraic methods transform the template decoder's output into a full key recovery attack. In Section 6 we present our results, and conclude with some discussion.

2. THE TEMPLATE-ALGEBRAIC ATTACK

The template-algebraic side channel attack [20] is a profiling attack which uses a constraint solver to find the optimal key given the aposteriori probabilities of the entire state of the DUT, and not just those of the individual key bytes.

A central component of the attack is a constraint solver, and more specifically a pseudo-Boolean optimizer, described in more detail in [1]. In contrast to traditional constraint solvers such as SAT solvers, the pseudo-Boolean optimizer also receives, along with the logical constraints which must be **satisfied**, an additional goal function which must be **minimized**. A pseudo-Boolean optimizer is generally slower and less efficient than a single-purpose SAT solver.

The Template-TASCA attack follows these steps:

1. In a first offline phase, 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] or template attacks [8]. In contrast to standard template attacks, which only profile the key bytes, this attack creates templates for many internal states of the DUT.

2. Next, in a second offline phase, the DUT is profiled and a decoding process is devised, in order to map between a single power trace and a vector of leaks. The output of the decoder in this phase is a full probability vector for each leak, listing the aposteriori probability of the leak having each possible value, conditioned on the specific trace being received. For example, in the case of a Hamming weight-based template decoder, each potential leak will have an associated vector of 9 aposteriori probabilities corresponding to Hamming weights 0 to 8.

3. After the offline phase, the attacker is provided with a small number of power traces. The traces are accompanied by **auxiliary information** such as known plaintext and ciphertext. The decoding process is ap-

plied to the power trace, and a vector of aposteriori probabilities for each leak is recovered.

4. The leak vector, together with a formal description of the algorithm implemented in the DUT, is represented as a system of equations. This equation set also includes any additional auxiliary information.

5. A constraint solver evaluates the equation set and attempts to find a candidate key which satisfies the encryption algorithm while maximizing the aposteriori probability of the state elements involved.

6. Optionally, some post-processing can be used in order to brute force the remaining key candidates provided by the solver. As shown in [20], the key can be recovered within a few days, even if as many as 5 key bytes returned by the solver are incorrect.

2.1 Related Work

As stated in [5], a constraint solver is a piece of software which tries to find an assignment over a set of variables such that a set of user-defined constraints is satisfied. Constraint solvers are widely used in the hardware design community for model checking, FPGA routing and other purposes. The first attempt to attack a modern cipher (DES[18]) using constraint solvers was published in 2000 by Massacci et al. in [16]. This attack methodology was named Logical Cryptanalysis or Algebraic Cryptanalysis. In Algebraic Cryptanalysis attacks, cryptosystems are represented as systems of equations. A constraint solver is then applied to find the cryptographic key satisfying these equations. In [23] and subsequent works Renauld et al. applied SAT solvers to the problem of side-channel analysis. This new attack methodology was named *Algebraic Side-Channel Analysis* (ASCA).

To carry out an ASCA attack, the attacker recovers a vector of **side-channel leaks** (such as Hamming weights or Hamming distances) from the power trace, then writes an **equation set** mapping these leaks to the evolution of the internal state of the device; finally, a **constraint solver** is used to find the secret key satisfying these equations. In [23] and [24] it was shown that if the side-channel vector is represented perfectly it is possible to recover the key from unprotected AES[12] and PRESENT[6] software implementations with very low data complexity (typically one or two power traces).

One drawback of the ASCA method is that it requires perfectly accurate side-channel data. The sensitivity of the ASCA methodology to noise or decoding errors in the side-channel leak vector severely limits its practicality. In [19] a new attack methodology called *Tolerant Algebraic Side-Channel Analysis (TASCA)* was presented. This methodology allows the algebraic methods of [24] to be used for key recovery from a very small amount of side-channel information, even in the presence of reasonable amounts of measurement noise. Another method of tolerating some noise was also introduced in [28], and a similar approach was also discussed in [17]. Finally, in [20], the TASCA method was adapted to work on outputs from a general template decoder. However, only the online phase of attack was modeled, based on simulated probability outputs of a hypothetical ideal decoder. In contrast, this work provides a complete evaluation of the attack, starting with actual traces and describing both the offline and the online phases.

Techniques based on the stochastic approach [25], principal components analysis [9], machine learning [10] and multivariate regression [13] were also shown to reduce the number of traces required for preprocessing. Another related work in the field is [2].

3. THE LEAK MODEL

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 intermediate states. The leakage information consists of the aposteriori probabilities $Pr\left(s_i|x\right)$. In the profiling phase we learn the power consumption behavior of the DUT. To this end we assume 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 aposteriori probabilities for the intermediate state bytes from the DUT.

3.1 Experiment Setup

The attacks were carried out on the IAIK WS2 data set, provided online[2] as part of the supplementary data to [14]. The data set consists of 200 power traces of the first round of AES, each with the same secret key but with different plaintexts. The device under test was an Atmel 8052-compatible micro-controller[11] running the standard 8-bit implementation of AES. A standard correlation power analysis[7] attack on these traces recovers the secret key using approximately 50 traces in the online phase.

The WS2 data set traces implement no countermeasures and are fully-aligned. Thus, at each point in time all 200 traces are completely synchronized and are performing exactly the same instruction.

3.2 Leaks of Information

As stated in [14], micro-controller implementations of AES are expected to leak the Hamming weights of the state bytes they process. Thus, we designed our the decoder to output the aposteriori probabilities for the 9 possible Hamming weights of each byte.

As described in [14], we assumed that the Hamming weight leakage of information in the power consumption is of the form $Power = Power\left(HW\left(s_i\right)\right) + noise$. $HW\left(s_i\right)$ is the Hamming weight of state byte s_i, and $noise$ 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 what a *naive Bayes* classifier relies on. Therefore we chose to use Matlab's NaiveBayes classifier for profiling and decoding purposes.

3.3 Leaks We Used

Our decoded was designed to output the following leaks:

• The *AddKey* and *AddRoundKey* operations leak the Hamming weights of the 16 state bytes after the XOR with the key/round key.

• The *SubBytes* operation is implemented as a look-up table (LUT), which leaks only the Hamming weights of the 16 state bytes after the *SubBytes* operation (and not any other internal state information).

• The *ShiftRows* operation is implemented logically as index shuffling and as such does not leak any information.

[2]URL: http://dpabook.org/onlinematerial/

• The *MixColumns* operation is implemented using 8-bit XTIME and XOR operations as specified in [12] and as such leaks 36 additional bytes of internal state per round. In addition to the 16 leaks of the final state, this gives a total of 52 leaks per subround.

In total, each round of AES leaks 84 Hamming weights of 8-bit values. Note that we assumed that the leaks from the key expansion process are not available to the attacker. In addition, since all traces used identical keys it was impossible to recover any direct information about the key bytes themselves.

4. PROFILING THE TEMPLATE ATTACK

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. In the WS2 data set we are given 200 traces, each of which consists of 100,000 samples. 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.

4.1 Methodology

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.

4.2 Finding Candidate Features

The first step of building the decoder was identifying the points in time where the traces contain 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) we divide the 200 traces into 9 Hamming weight classes (0 through 8) according to the true value the leak takes.

Notice 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 eight values for Hamming weight 1 and eight values for Hamming weight 7. This fact, combined with the small size of our training set, meant there were only a few, if any, traces with leaks corresponding to those Hamming weight values. For this reason we merged classes 0 and 1 into class 2, and classes 8 and 7 into class 6.

After dividing our 200 traces into the 5 classes mentioned above, we needed to build a classifier that can distinguish between traces of these classes. Similar to what was done in [22], 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 [22]: instead of taking sequential samples from a trace as features, we take 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 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.

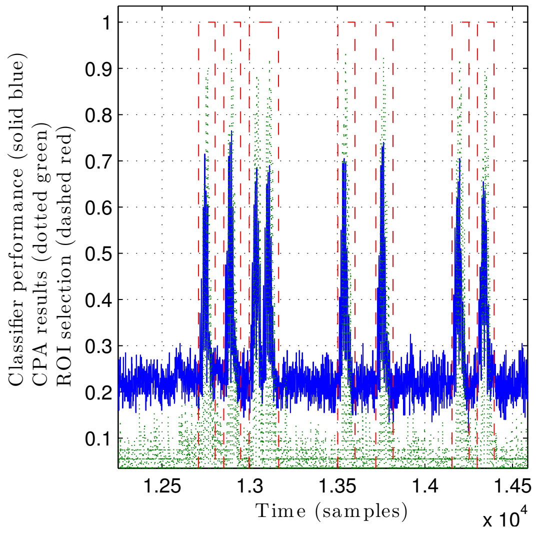

Figure 1: Three-featured classifier feature selection. Regions of interest are indicated by the dotted line.

4.3 Improving Candidate Search Efficiency

On our system, the 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 [7]: 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 previous subsection only in regions of interest. Since the regions of interest are about typically less than 2% of the entire trace, this speeds up the search process dramatically – from 15 minutes down to approximately 20 seconds per leak.

4.4 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 $used\_features$. $used\_features(c) = True$ 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 discarded in this process. The last trained classifier on a group of 180 traces is saved as the best classifier for the leak.

4.5 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 estimation for classes 0 through 8. In addition, the very small amount of training data provided to 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. 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

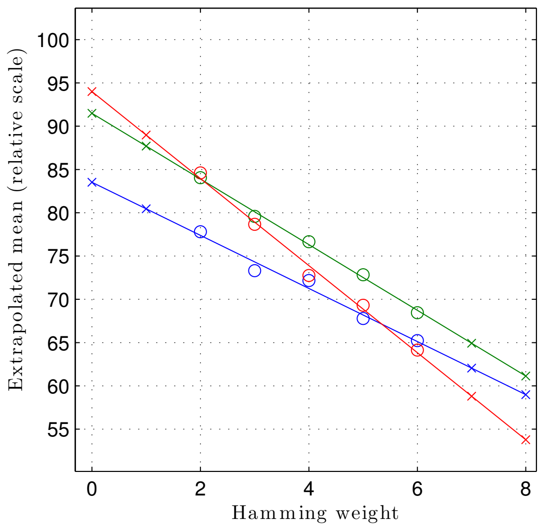

Figure 2: An example of linear regression of the mean values of 3 features. Circles indicate means output by the Naive Bayes classifier. Crosses indicate means recomputed via linear regression for classes 0 through 8.

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

4.6 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 will give us for each trace an aposteriori 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 aposteriori 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 the correct leaks. Ideally we would like the correct leak value to be ranked 1, i.e. being given the highest probability by our decoder. We have 84 leaks, and 200 traces which gives us $84 \times 200 = 16800$ aposteriori estimations for the leaks. In

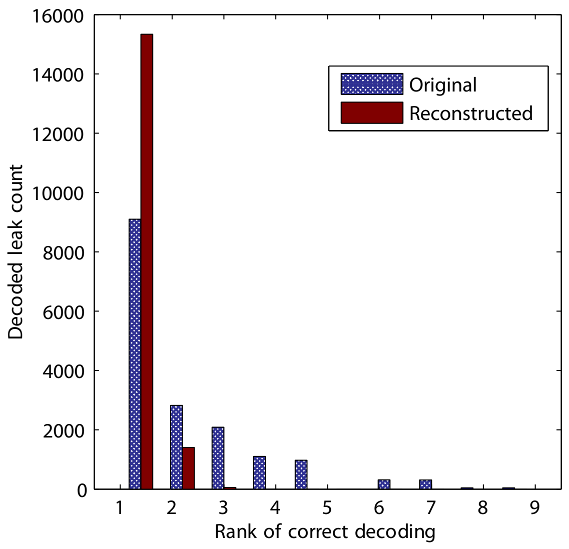

Figure 3: Ranks of correct leak values given by the classifiers. In hatched blue are the ranks given by the 5-classes original classifiers. In solid red are the ranks given by the 9-classes reconstructed classifiers.

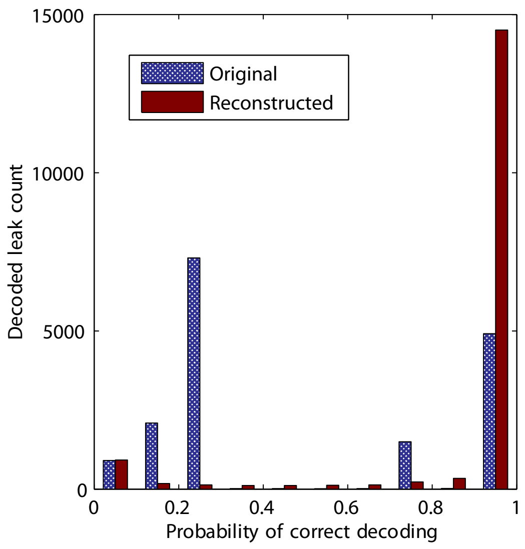

Figure 4: Probability of correct decoding by the classifiers. In hatched blue is the performance of the 5-classes original classifiers. In solid red is the performance of the 9-classes reconstructed classifiers.

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. The reconstructed classifiers (with regressed mean values) dramatically improved the given estimations for the correct states.

5. THE ALGEBRAIC STAGE

Given a single trace, the output of the template decoder's online stage is a matrix consisting of 84 rows, each containing 9 aposteriori 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[26]. However, as stated in Section 1.1, not all combinations of state bytes correspond to a valid encryption. Instead, we employ a constraint solver to find the valid states which maximizes the aposteriori probability for all state bytes simultaneously.

The solver we chose to use is SCIPspx version 1.2.0 [3, 4, 1]. SCIPspx won the first prize for non-linear optimizer in the Pseudo-Boolean Evaluation Contest of SAT 2009 [15]. 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 [27]. The solver was run on 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.

5.1 An equation set for AES

The AES instances we submitted to the solver were created according to the principles described in [20], and are similar in form to the example found in the appendix of [20]. The AES equation set as described in [20] expects to receive aposteriori probabilities for the Hamming weights of the key bytes 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 key bytes themselves. Instead, we used the apriori distribution for Hamming weights of an 8-bit value for the key bytes:

$$\Pr(HW = x) = \frac{\binom{8}{x}}{256} \tag{5}$$

Some of our experiments were performed on a single trace in the offline phase, while some 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.

5.2 Combinatorial Exclusion

Since the decoder used in the template phase estimates each potential value by a Gaussian random variable, the aposteriori 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

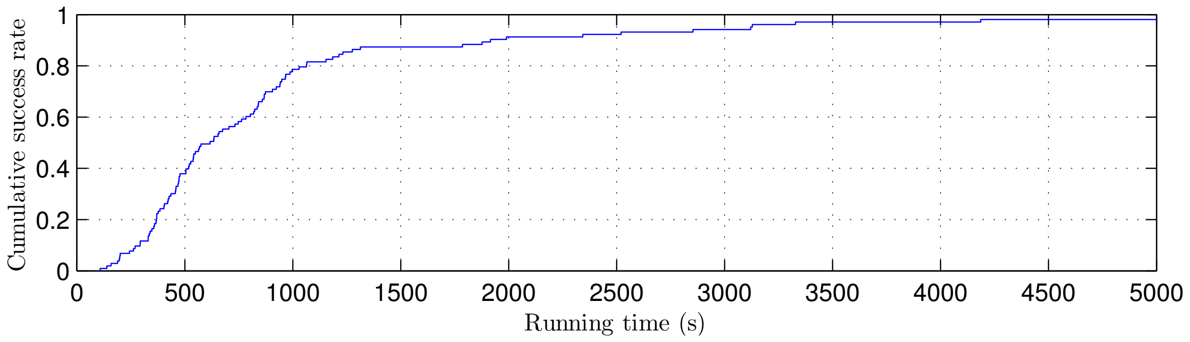

Figure 5: Running time distribution for the two-trace scenario

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 aposteriori probabilities of all values which are far less likely than the least likely correct value. In our data set the most unlikely value had an aposteriori probability of $3 \cdot 10^{-40}$. 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 aposteriori probability matrix.

6. RESULTS

6.1 Conditions for success

As stated in [20], 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 aposteriori 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.

6.2 Double-trace attack

Our first attack was performed on pairs of traces, which were combined as described in Section 5.1. 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.

6.3 Single-trace attack

Our second set of attacks was carried out with the lowest possible online data complexity - a single trace per attack. Due to limited time we were not able to apply our attack to a large set of single traces. Our preliminary results, however, are very promising.

Out of the 13 instances we evaluated, 3 instances returned all 16 bytes of the correct key and 7 returned a value which is within brute-force distance from the correct key (either 2, 3 or 4 incorrect key bytes). The median running time of the single-trace instances was 24 hours, while the maximum time was 59 hours.

7. DISCUSSION

7.1 Comparison with Solver-Based Attacks

As stated in [20], 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, since a solver does not have any efficient way of representing the objective function which contains the aposteriori 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 [20], 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, both in the double-trace and in the single-trace scenarios. In the single-trace scenario the solver-based approach had 0% success in finding the correct key, in comparison to a 75% success rate for the optimizer-based method. In the double-trace scenario 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.

7.2 Conclusion and Future Work

This report shows how the Template-TASCA approach first described in [20] 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. One possible approach would be to use the similarity between different template decoders such as the 16 decoders which extract the 16 different *AddKey* state bytes – it may be possible to train one decoder on all 16 leaks, allowing the data complexity to be reduced even further. Another promising option would be to replace the template component of the attack with alternative profiling-based methods. Finally, it would be interesting to gauge the effectiveness of the Template-TASCA approach on devices protected by various countermeasures such as masking.

Acknowledgements We thank the anonymous reviewers for their time and their helpful comments.

8. REFERENCES

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

[2] A. Bechtsoudis and N. Sklavos. Side channel attacks cryptanalysis against block ciphers based on fpga devices. In *VLSI (ISVLSI), 2010 IEEE Computer Society Annual Symposium on*, pages 460–461. IEEE, 2010.

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

[4] T. Berthold, S. Heinz, M. E. Pfetsch, and M. Winkler. SCIP – Solving Constraint Integer Programs. SAT 2009 competitive events booklet, 2009.

[5] J. Biskup, D. M. Burgard, T. Weibert, and L. Wiese. Inference control in logic databases as a constraint satisfaction problem. In *Information Systems Security*, pages 128–142. Springer, 2007.

[6] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In P. Paillier and I. Verbauwhede, editors, *CHES*, volume 4727 of *LNCS*, pages 450–466. Springer, 2007.

[7] E. Brier, C. Clavier, and F. Olivier. Correlation power analysis with a leakage model. In M. Joye and J.-J. Quisquater, editors, *CHES*, volume 3156 of *LNCS*, pages 16–29. Springer, 2004.

[8] S. Chari, J. R. Rao, and P. Rohatgi. Template Attacks. In B. S. K. Jr., Çetin Kaya Koç, and C. Paar, editors, *CHES*, volume 2523 of *LNCS*, pages 13–28. Springer, 2002.

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

[10] A. Heuser and M. Zohner. Intelligent machine homicide. In *Constructive Side-Channel Analysis and Secure Design*, pages 249–264. Springer, 2012.

[11] IAIK IMPA Lab. IAIK IMPA Lab Infrastructure. Online.

[12] 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.

[13] Y. Kim, N. Homma, T. Aoki, and H. Choi. Security evaluation of cryptographic modules against profiling attacks. In T. Kwon, M.-K. Lee, and D. Kwon, editors, *ICISC*, volume 7839 of *LNCS*, pages 383–394. Springer, 2012.

[14] S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards (Advances inInformation Security)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[15] V. Manquinho and O. Roussel. Pseudo-Boolean Competition 2009. Online, July 2009.

[16] F. Massacci and L. Marraro. Logical Cryptanalysis as a SAT Problem. *J. Autom. Reason.*, 24(1-2):165–203, 2000.

[17] M. S. E. Mohamed, S. Bulygin, M. Zohner, A. Heuser, M. Walter, and J. Buchmann. Improved algebraic side-channel attack on AES. In *HOST*, pages 146–151. IEEE, 2012.

[18] National Institute of Standards and Technology. *FIPS PUB 46-3: Data Encryption Standard (DES)*. Oct. 1999.

[19] Y. Oren, M. Kirschbaum, T. Popp, and A. Wool. Algebraic Side-Channel Analysis in the Presence of Errors. In S. Mangard and F.-X. Standaert, editors, *CHES*, volume 6225 of *LNCS*, pages 428–442. Springer, 2010.

[20] Y. Oren, M. Renauld, F.-X. Standaert, and A. Wool. Algebraic side-channel attacks beyond the hamming weight leakage model. In E. Prouff and P. Schaumont, editors, *CHES*, volume 7428 of *LNCS*, pages 140–154. Springer, 2012.

[21] D. Oswald and C. Paar. Breaking mifare desfire mf3icd40: Power analysis and templates in the real world. In B. Preneel and T. Takagi, editors, *CHES*, volume 6917 of *LNCS*, pages 207–222. Springer, 2011.

[22] C. Rechberger and E. Oswald. Practical template attacks. In C. H. Lim and M. Yung, editors, *WISA*, volume 3325 of *LNCS*, pages 440–456. Springer, 2004.

[23] M. Renauld and F.-X. Standaert. Algebraic Side-Channel Attacks. In F. Bao, M. Yung, D. Lin, and J. Jing, editors, *Inscrypt*, volume 6151 of *LNCS*, pages 393–410. Springer, 2009.

[24] M. Renauld, F.-X. Standaert, and N. Veyrat-Charvillon. Algebraic Side-Channel Attacks on the AES: Why Time also Matters in DPA. In C. Clavier and K. Gaj, editors, *CHES*, volume 5747 of *LNCS*, pages 97–111. Springer, 2009.

[25] W. Schindler, K. Lemke, and C. Paar. A stochastic model for differential side channel cryptanalysis. In J. R. Rao and B. Sunar, editors, *CHES*, volume 3659 of *LNCS*, pages 30–46. Springer, 2005.

[26] N. Veyrat-Charvillon, B. Gérard, M. Renauld, and F.-X. Standaert. An optimal key enumeration algorithm and its application to side-channel attacks. In L. R. Knudsen and H. Wu, editors, *Selected Areas in Cryptography*, volume 7707 of *LNCS*, pages 390–406. Springer, 2012.

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

[28] X. Zhao, T. Wang, S. Guo, F. Zhang, Z. Shi, H. Liu, and K. Wu. SAT based Error Tolerant Algebraic Side-Channel Attacks. 2011 Conference on Cryptographic Algorithms and Cryptographic Chips (CASC2011), July 2011.