Noname manuscript No.
(will be inserted by the editor)

Side-Channel Cryptographic Attacks using pseudo-Boolean Optimization

Yossef Oren · Avishai Wool

Received: date / Accepted: date

Abstract Symmetric block ciphers, such as the Advanced Encryption Standard (AES), are deterministic algorithms which transform *plaintexts* to *ciphertexts* using a *secret key*. These ciphers are designed such that it is computationally very difficult to recover the secret key if only pairs of plaintexts and ciphertexts are provided to the attacker. Constraint solvers have recently been suggested as a way of recovering the secret keys of symmetric block ciphers. To carry out such an attack, the attacker provides the solver with a set of equations describing the mathematical relationship between a known plaintext and a known ciphertext, and then attempts to solve for the unknown secret key. This approach is known to be intractable against AES unless side-channel data – information leaked from the cryptographic device due to its internal physical structure – is introduced into the equation set.

A significant challenge in writing equations representing side-channel data is measurement noise. In this work we show how casting the problem as a pseudo-Boolean optimization instance provides an efficient and effective way of tolerating this noise. We describe a theoretical analysis, connecting the measurement signal-to-noise ratio and the tolerable set size of a non-optimizing solver with the success probability. We then conduct an extensive performance

Y. Oren
Network Security Lab, Computer Science Department, Columbia University in the City of New York, 1214 Amsterdam Avenue, New York, NY 10027, USA

A. Wool
Cryptography and Network Security Lab, School of Electrical Engineering, Tel Aviv University, Ramat Aviv 69978, Israel

Y. Oren
E-mail: yos@cs.columbia.edu

A. Wool
E-mail: yash@eng.tau.ac.il

evaluation, comparing two optimizing variants for dealing with measurement noise to a non-optimizing method. Our best optimizing method provides a successful attack on the AES cipher which requires surprisingly little side-channel data and works in reasonable computation time. We also make available a set of AES cryptanalysis instances and provide some practical feedback on our experience of using open-source constraint solvers.

Keywords Application paper · Cryptanalysis · pseudo-Boolean optimizers · Side-channel attacks

1 Introduction

1.1 Background

Block ciphers are a common cryptographic building block used in many secure applications and protocols. A block cipher receives as input a plaintext and a key, and outputs a ciphertext which depends on the plaintext and the key. Due to the internal structure of a block cipher, it is generally very simple to calculate a ciphertext, given a plaintext and a key. It is also simple to calculate the plaintext, given the ciphertext and a key. It is, however, considerably more difficult to calculate the key, given one or more pairs of plaintext and ciphertexts. This difficulty stems from the internal structure of most block ciphers (described further in Section 2.1) which makes it difficult to write a simple, low-degree Boolean equation describing the key as the function of the plaintext and the ciphertext. The field of *block cipher cryptanalysis* deals with discovering and describing attacks on block ciphers, including efficient ways of recovering the secret key of a block cipher from plaintext-ciphertext pairs.

As the field of constraint solvers developed, it became interesting to check whether a constraint solver can be used directly for cryptanalysis. To carry out such an attack, the attacker writes the relatively simple equation set which describes the ciphertext as the function of the plaintext and the key, then fixes the values of the plaintext and ciphertext, and finally uses a solver to try to find an assignment to the key bits which satisfies the equation set. The first attempt to attack a modern cipher – the Data Encryption Standard (DES [21]) – using constraint solvers was published in 2000 by Massacci et al. in [16]. This attack methodology was named logical cryptanalysis or algebraic cryptanalysis. Massacci et al.'s work and subsequent followups ([30, 18, 11, 7]) consistently demonstrated that modern cryptographic algorithms cannot be directly attacked by constraint solvers.

In parallel to classical cryptanalysis, which treats the block cipher as an abstract mathematical algorithm, another field of research tried to attack block ciphers based on the physical properties of their actual implementations. This discipline is called side-channel cryptanalysis, and was first described in an academical setting by Kocher et al. in [12][1]. As formally defined in [14], side-

[1] It is believed that this form of attack was well known to the signals intelligence community from as early as WWII.

channel attacks are attacks which reveal the secret keys of cryptographic devices by observing their physical properties. The work of [12] and others has shown that the physical properties of a cryptographic device (such as its temperature, its electromagnetic emanations, and so on) depend on the secret key, and that it is thus possible to extract the key by processing this information leakage.

Algebraic Side-Channel Analysis (ASCA), first described in [28], combines the two fields of algebraic cryptanalysis and side-channel cryptanalysis. A standard algebraic cryptanalysis instance contains equations describing the actual encryption algorithm. An ASCA instance contains additional equations representative of the physical emanations caused by this algorithm's execution. 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 [28] and [27] it was shown that if the side-channel vector is represented perfectly it is possible to recover the key from unprotected software implementations of the AES [20] and PRESENT [5] ciphers with very low data complexity (typically one or two power traces).

The advantage of the ASCA method is its very low data complexity. However, the ASCA methodology is very sensitive to noise or decoding errors in the side-channel leak vector, severely limiting its practicality. In [22] a new attack methodology called *Tolerant Algebraic Side-Channel Analysis (TASCA)* was presented. This methodology allows the algebraic methods of [27] 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 modification to the standard ASCA attack which can also tolerate some noise, called Set-ASCA, was introduced in [31]. The performance of the methods of [27] and [31] is the focus of this paper.

1.2 Contributions

In this paper we show how the TASCA or Set-ASCA methodology can be used to recover secret keys from cryptographic devices even if the data available to the attacker is limited both in quantity (data complexity) and in quality (signal to noise ratio). We define the *set size* k as a parameter describing the error-tolerance of our attack and provide a theoretical analysis which connects the measurement signal-to-noise ratio and the set size with the success probability. Comparing TASCA and Set-ASCA, we show that optimizing solvers have a distinct advantage over non-optimizing solvers in our scenario. Our results show that using the TASCA method the secret key can be recovered from 60%-70% of AES instances, even when only a single power trace is provided, and even when 20% of the trace signal is corrupted by noise. This new cryptanalytic capability may compromise secure systems whose defense against (statistical) side-channel attacks is an aggressive re-keying schedule which results in a small

amount of traces per given key. We also make available a rich and interesting set of instances and some practical feedback on our experience using open-source constraint solvers, both of which may be of use to the developers of these solvers.

The rest of the paper is organized as follows: Section 2 provides the reader with fundamental information about block ciphers and side-channel attacks. It then describes various approaches to algebraic side-channel attacks, and in particular the TASCA method of [22]. In section 3 we provide a formal description of the problem statement and of the goal function. In section 4 we describe a theoretical analysis, connecting the measurement signal-to-noise ratio and the set size in a Set-ASCA or TASCA solver with the success probability. Next, in Section 5, we describe an experiment setup, both in terms of the device under test (DUT) and of the software and hardware configuration of the solver. In Section 6 we list results obtained using standard (non-optimizing) ASCA on our simulated AES implementation. In Section 7 we describe the actual tolerant attack on AES and its performance, and we conclude with some discussion in Section 8.

1.3 Related Work

In [28] and subsequent works Renauld et al. applied SAT solvers to the problem of side-channel analysis. Interestingly, while straightforward cryptanalysis was consistently shown to be beyond the capability of solvers, once equations based on side-channel information were introduced into the equation set the situation changed. Renauld et al. showed how a solver-based attack on the modern cipher AES can terminate in less than a minute with the correct key when provided with suitable side-channel information. This new attack methodology was named *Algebraic Side-Channel Analysis* (ASCA). However, the ASCA method requires perfectly accurate side-channel data.

One method for dealing with measurement noise was introduced in [27] and more recently investigated in [31]. In this approach, which we call *Set-ASCA*, each entry in the recovered leak vector is not represented as an equation allowing a single acceptable value. Instead, the equations accept any value from a *set* of several possible values. The values in the set are each equally acceptable, that is, there is no incentive for the solver to choose one value over another. The resulting equation set is then submitted to a standard SAT solver. As shown in [31], this approach is quite satisfactory given that enough leaks and auxiliary information are provided to the solver.

In [22] a different method for dealing with noise is described, called *Tolerant Algebraic Side-Channel Analysis* (TASCA). The main ingredient in [22] is the use of an optimizing solver, with a goal function that minimizes the amount of modeled noise. In this work we evaluate the TASCA approach and compare it to the Set-ASCA approach.

2 Preliminaries

2.1 Symmetric Block Ciphers

This paper deals with solver-based attacks on symmetric block ciphers, one of the fundamental building blocks of modern cryptography. As defined in [17, §7.2.1], a block cipher is an *invertible function* which maps *plaintext* blocks to *ciphertext* blocks. In most block ciphers the plaintext and ciphertext blocks are of equal size, called the *block size* n. The block cipher is also parameterized by a secret key K with a *key size* of k bits. For n-bit plaintext and ciphertext blocks and a fixed key, the encryption function is a *bijection*, defining a permutation on n-bit vectors, with each key potentially defining a different bijection. The key may have a different size than the block size and its value is typically chosen at random.

Many block ciphers follow an *iterated design*, involving the sequential repetition of an internal building block called a *round function*. As illustrated in Figure 1, the round function is repeated r times, and each invocation of the round function is supplied with a round key rk_i derived from the input key k by invoking an invertible key derivation function. The intermediate values of the iterative cipher are called the *state bits*, with the initial state equal to the plaintext and the final state equal to the ciphertext.

2.1.1 The Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) cipher is a block cipher proposed by Joan Daemen and Vincent Rijmen in [8], and adopted by the U.S. government in [20] as an encryption algorithm for protecting sensitive government information. Since its introduction in 2001, AES has arguably become the world's most commonly used and well studied block cipher.

AES is an *iterated block cipher* with a block size n of 128 bits (16 bytes) – it inputs 128-bit plaintexts, has a 128-bit intermediate state (commonly treated as 16 state bytes of size 8 bits each, arranged in 4 rows by 4 columns), and produces 128-bit ciphertexts. AES is defined with three different security levels, with each level specifying a different key size k and round count r. The most common variant of AES is AES-128, which has a key size k of 128 bits and a round count r of 9 rounds. In all variants of AES the *key derivation function* expands the master key into the appropriate number of round keys which are always 128 bits in length.

Each round in AES consists of the following four elementary operations, or subrounds, which are performed in series:

- **SubBytes** - in this subround each byte of the state is substituted with another byte according to a 256-entry substitution table. The AES substitution function, commonly called the S-box, is defined as taking a multiplicative inverse of each state byte in $GF\left(2^8\right)$, followed by an affine transformation over $GF\left(2\right)$.

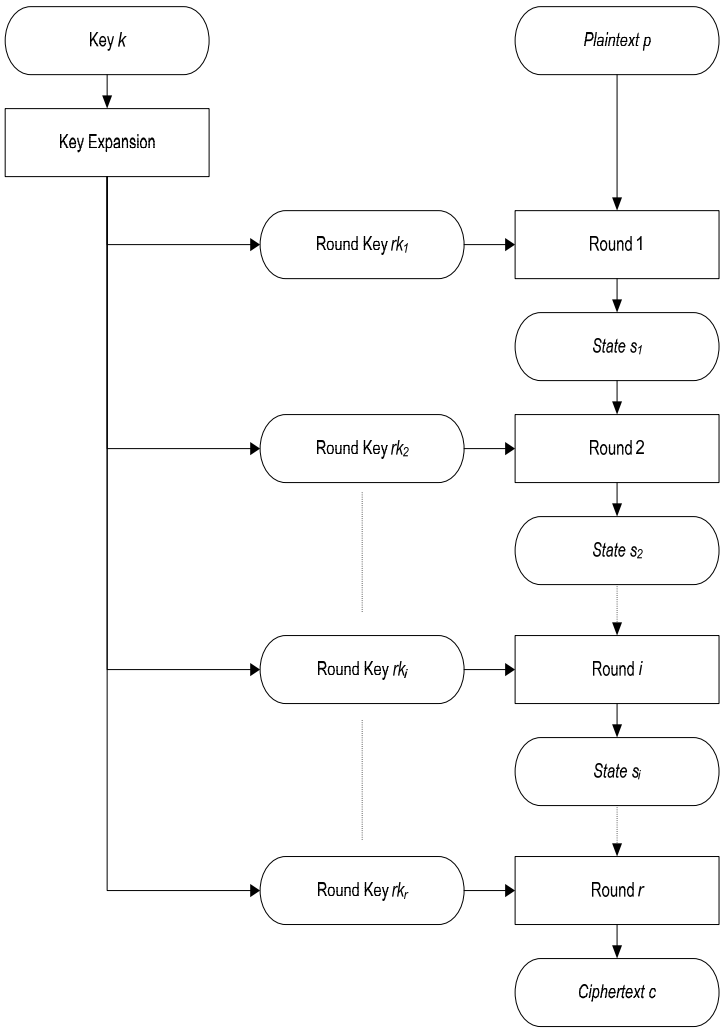


Fig. 1 General structure of an iterated block cipher

- **ShiftRows** - in this subround the state is split into 4 rows of 4 state bytes each, and each row of the state is cyclically shifted by a differing amount of bytes.
- **MixColumns** - in this subround the state is split into 4 columns of 4 state bytes each, and an affine transform is applied to each column, by treating the column as a polynomial over in $GF\left(2^8\right)$ and multiplying it by another fixed polynomial.
- **AddRoundKey -** in this subround the current state is combined with the appropriate round key using a bitwise exclusive or (XOR) operation.

Round		1				2					9				10		
Subround	1	2	3	4	5	6	7	8	9	···	34	35	36	37	38	39	40
Operation	K*	B	R	M	K	B	R	M	K	···	B	R	M	K	B	R	K

Table 1 The subround structure of AES-128 (K*=AddKey, B=SubBytes, R=ShiftRows, M=MixColumns, K=AddRoundKey)

A complete AES-128 encryption begins with a special **AddKey** subround, in which the plaintext is mixed with the original cipher key, (and not one of the derived round keys) followed by 9 rounds as described above, and terminating with a final round which consists only of SubBytes, ShiftRows and AddRoundKey. Thus, the complete AES-128 encryption operation consists of 40 subround operations which transform the cipher state from plaintext to ciphertext, as illustrated in Table 1.

2.2 Side-channel attacks

As formally defined in [14], side-channel attacks are attacks which reveal the secret keys of cryptographic devices by observing their physical properties. The work of [12] and others has shown that the physical properties of a cryptographic device (such as its power usage, its electromagnetic emanations, and so on) depend on the secret key, and that it is thus possible to extract the key by processing this information leakage. In the past 25 years since they have been brought to use in an academic setting, side-channel attacks have been effectively and consistently used to attack systems which are provably secure in theory, due to information leaked by the physical aspects of their implementation.

This paper deals with a specific type of side-channel leakage called the power side-channel. The power side-channel is obtained by precisely measuring the instantaneous power consumption of the DUT as it performs a cryptographic operation, then using this leakage to learn about the internal state of the DUT. As stated in [14], cryptographic hardware typically uses a CMOS fabrication process. CMOS-based registers (or flip-flops) consume more power if they switch between logic states, and less power if they remain in the same state. Thus, the instantaneous power consumption of a DUT is correlated with the number of device registers changing state in a particular time period, or the **Hamming distance** between two consecutive hardware states. If more information about the hardware is available, it is possible to construct more elaborate leakage models which capture additional information about the DUT. For a discussion on the use of different leakage models for our attack, see [23].

The specific device we target in this article is an 8-bit micro-controller running a software implementation of AES. As shown in [14], this class of microcontroller can only process data in chunks of 8 bits at a time. Every time the microcontroller loads such an 8-bit data element into its logic unit for processing, it first resets the logic unit to a default value (typically 0). This

means that the amplitude of the power trace is correlated to the **Hamming weight** of the byte currently being processed, where the Hamming weight is defined as the amount of bits in the byte which are set to 1. To recover a secret key from power traces, the attacker typically measures the amplitude of the power trace at one or more "interesting" points in time. Next, the attacher applies a *decoding process* to these measurements to arrive at estimated values for the Hamming weights of various bytes processed by the microcontroller during the cryptographic operation. Finally, a variety of methods from various disciplines of signal processing, statistics and machine learning, as described in detail in [14], may be used to arrive from these estimated measurements to the actual secret key bits.

2.3 Algebraic Side-Channel Attacks

First presented in [28], algebraic side-channel attacks attempt to use constraint solvers to perform side-channel attacks. To do so, an attacker writes an equation with both the actual encryption algorithm and a representation of the physical emanations caused by this algorithm's execution. 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 general, the equation set used in an algebraic side-channel attack consists of three sets of equations: those that describe the encryption process (e.g. the evolution of the state bytes from plaintext to ciphertext as a function of the key), those that describe the measurement process (e.g., what is the relation between the measured side-channel leaks and the encryption state bytes), and those that describe the results of the physical measurement itself. While the first two equations are completely deterministic, the actual measurement results are subject to the physical limitations of the device under test and of the measurement setup, and are as such sometimes corrupted with errors. There are several approaches of dealing with this measurement error. We describe them below, and contrast them in more detail in the remainder of the paper:

- In the conventional algebraic side-channel attack (ASCA) approach [27, 28], only **the most probable measurement** is considered to satisfy the equation set. The advantage of this method is its simplicity and high speed, but this method is highly sensitive to measurement errors and requires a very high signal-to-noise ratio which is often impractical.
- The Set-ASCA approach, first suggested by [27] and later expanded in [31], allows **several of the more probable measurements** to satisfy the equation system. The set of satisfying assignments is chosen according to some heuristic, such as the k most likely values or all values whose **a posteriori probability** is greater than some threshold. A similar method was also explored in [19].

– The Tolerant ASCA (TASCA) approach also allows several measurements to satisfy the equation system. However, this method also adds a **goal function** which indicates which measurements are more likely than others, based on confidence data output while decoding the physical measurement. This effectively transforms the problem from a **satisfiability** problem to an **optimization** problem, guiding the constraint solver toward the most probable input value.

ASCA and Set-ASCA equation sets are classical satisfiability instances, without a goal function. The solver applied to them in [27,28] and [31] was CryptoMiniSAT [1], a solver which is well adapted to deal with cryptographic problems, as XOR operations (very frequent in cryptographic algorithms) are managed by the solver using specific optimized clauses. TASCA equation sets form an optimization instance and include a goal function. These instances were solved using the SCIP constraint solver [4].

3 Constraint Programming Representation

3.1 Motivation

In a TASCA attack we assume that the attacker is provided with a **device under test** which performs a cryptographic operation (e.g. encryption). While encrypting, the device emits a measurable **side-channel trace**, specifically a **power trace**, which is captured by an oscilloscope. A certain amount of **leaks** are modulated into the trace, due to the physical characteristics of the device under attack. These leaks can teach an attacker about the internal state of the DUT during various stages of the cryptographic operation. While the leaks are typically integral values (commonly **Hamming weights** or **Hamming distances**), the power trace itself is a continuous analog signal which exhibits **noise** or **errors** due to interference and to limitations of the capture mechanism (see [22, §1.2]).

The TASCA methodology uses the following steps to recover the secret key from a power trace:

1. In an offline phase, the DUT is first analyzed in order to identify **potential leaks**, for instance by reverse engineering.
2. Next, the DUT is profiled and a **decoding process** is devised. The decoding process extracts a vector of **leaks** from a power trace.
3. After the offline phase concludes, the attacker is provided with a few power traces (typically a single trace). The traces may also be accompanied by some **auxiliary information** such as the known plaintext/ciphertext associated with this trace. A special decoding process, described in more detail in [24], can be used to process the power consumption trace of the DUT and output a vector of Hamming weights corresponding to a certain cryptographic operation. This vector of decoded Hamming weights, which may contain errors due to the limitations of the decoding process, represents each leak as an integer value between 0 and 8.

4. The leak vector, together with a formal description of the DUT found through reverse engineering, is represented as a system of pseudo-Boolean equations. This equation set also includes the auxiliary information. The equation set is specially formed such that an optimizing solver can receive a leak vector which is a slightly different from the original vector (due to the effect of noise) and still find the correct key assignment. The equation set also contains a **goal function**, which is used by the solver to measure the quality of each candidate solution. In our specific case the goal function indicates that less-errored solutions are preferable to errored ones.
5. Finally, a solver such as SCIP [4] evaluates the equation set and attempts to find a candidate key which satisfies the equation set while minimizing the goal function. The solver may fail to terminate in a tractable time, or otherwise return a candidate key.
6. The candidate key (or, as stated in Subsection 3.4, its immediate neighborhood) is verified.

As indicated in the above list, there are several conditions which must all hold true before a TASCA attack succeeds. First, the **decoder** should succeed in extracting the leaks from the power trace with an error rate which the solver is robust enough to handle. Next, the **solver** should return some key, and not run for an intractable time. Finally, the returned key should be the **correct key**.

3.2 Formal Constraint Model

We created a TASCA equation set which represents an algebraic side-channel attack as a constraint optimization instance. The equation set consists of the following four sections:

1. **A general description of the cryptographic algorithm as a set of equations**: The cryptosystem is described by writing down internal state transformations leading from plaintext to ciphertext. The specification is very hardware-minded, with each state bit/memory element (flip-flop) typically represented as a sequence of variables representing its evolution in time, and each combinational element (gate) finding its way into an equation connecting the variables. The specific encoding chosen for AES is described in more detail below.
2. **An assignment of any known inputs to the algorithm:** These can be known plaintext or ciphertext, or even more subtle hints such as the relationship between two consecutive unknown plaintexts.
3. **A specification of the measurement setup:** The actual side-channel measurement is mapped to the internal state according to the structure of the physical hardware device. For example, an 8-bit microcontroller-based implementation will typically leak the Hamming weight of individual state bytes as they are accessed, while a parallelized ASIC will typically leak the Hamming distance between the former and present values of all bits in the

device's internal state. It should be noted that when attacking the same cipher running on different target architectures, the measurement setup is usually the only section of the equation set which needs to be modified.

4. **A set of potentially errored measurements:** This section matches the measurements described in the previous section to actual outputs of the estimation phase. As stated previously, the main point of the TASCA approach is to allow errors in the estimation. This is done in our model by adding additional *error variables* to the above-mentioned measurement equations. These error variables are used to cancel out errors in the measurements. This section is the only part of the equation set which tolerates errors, as the equations all other sections are precisely defined, and it only accounts for 1% to 5% of the entire set of equations for the cryptosystems we tested. A more generic approach, based on detailed feedbacks from the physical decoding phase, is explored in [23].

The representation language we chose is the OPB notation used by the pseudo-Boolean solver SCIP [4]. The formal mathematical model, and how we chose to encode it as a constraint programming problem, is described below. A reference instance constructed using this model is included in the Appendix.

3.2.1 Decision Variables

As stated in Subsection 2.1.1, the internal state of the AES cipher is stored in 128 **state bits**, which are commonly grouped into 16 8-bit **state bytes**. These state bytes are initially loaded with the plaintext to be encrypted. During the encryption process, the state bytes are mixed with the **round key bytes** and are further manipulated by a series of 10 **cipher rounds**, each of which modifies the cipher's 128-bit internal state. At the end of the last encryption round the state bytes contain the **ciphertext**, which is the output of the encryption process. The 10 rounds can be further broken down into **40 subrounds**, as described in Table 1. There are four general subround types: AddKey/AddRoundKey, SubBytes, ShiftRows and MixColumns.

To model the TASCA instance as a constraint problem, we created several sets of binary variables:

- **State variables** $S_{t,i,j}$, which describe the evolution of the cipher state over time
- **Key variables** $K_{i,j}$ and $RK_{r,i,j}$, which represent the cipher key and the round keys derived from this key
- **Error variables** $e^{+}_{t,i}$ and $e^{-}_{t,i}$, which allow the solver to cancel out the errors inherent in the side-channel measurement process.

The **state variables** $S_{t,i,j}$ follow the evolution of the cipher state between subrounds. The binary variable $S_{t,i,j}$ corresponds to the value of bit j of state byte i at subround t, where $t \in [0, 41], i \in [0, 15], j \in [0, 7]$. We use the shorthand form $S_{t,i}$ to refer to the entire 8 bits of state byte i at subround t, and the shorthand form S_t to refer to the entire 16 bytes, or 128 bits, of the state

at subround t. Thus, the set of 128 binary variables S_0 represents the plaintext, the set of 128 binary variables S_{41} represents the ciphertext, and all other intermediate states represent the values of the 16 state bytes as they are transformed by the various subround operations. The following subsections describe in more detail how we realized each subround operation as an equation set.

The **key variables** $K_{i,j}$ and $RK_{r,i,j}$ represent the 128-bit key used by the AES-128 operation, as well as the 10 additional **round keys** derived from this key during the AES-128 key expansion phase. The binary variable $K_{i,j}$ corresponds to the value of bit j of secret key byte i, while the binary variable $RK_{r,i,j}$ the value of bit j of byte i of the round key for round r, where $r \in [0, 9], i \in [0, 15], j \in [0, 7]$. We use a shorthand form to represent these variables as well, using $RK_{r,i}$ to refer to the entire 8 bits of state byte i at round r and RK_r to refer to the entire 16 bytes, or 128 bits, of the round key for round r. We also use the notation K_i to refer to the entire 8 bits of key byte i, and K to refer to all 16 bytes, or 128 bits, of the key.

The **side-channel measurement error variables** $e^+_{t,i}$ and $e^-_{t,i}$ allow the constraint solver to overcome a limited amount of measurement noise in the side-channel leak equations. The side-channel measurements we consider in this work are the Hamming weights of individual 8-bit data elements, such as state bytes, key bytes or intermediate values, as they are processed by a microcontroller. The **actual** value of these side-channel leaks (that is, the value which would have been measured by an optimal decoder in an error-free case) is thus the integer sum of all the bits in the operand currently being processed by the microcontroller, or $\sum_{j=0}^{7} S_{t,i,j}$. The **measured** value $M_{t,i}$, which is returned by a non-optimal decoder being influenced by errors, may differ from the actual value due to the effect of noise. In this work we allow the actual value to deviate from the measured value by ± 1. To capture this error model, we add a pair of binary variables, $e^+_{t,i}$ and $e^-_{t,i}$, to each measurement equation, and use the goal function to instruct the solver to use as few of these error variables as possible. The final measurement equation for a single state byte is thus:

$$\sum_{j=0}^{7} S_{t,i,j} + e^+_{t,i} - e^-_{t,i} = M_{t,i}$$

As we note below, side-channel measurement equations exist not only for the state bytes, but also for the key bytes, the round key bytes, and for several additional intermediate values. A pair of error variables therefore exists for each of these side-channel measurements.

In addition to these primary variables, the AES instance also contains additional helper variables to hold intermediate values. As indicated in more detail below, some of these helper variables describe actual intermediate steps used by the 8-bit implementation of AES, while other variables are artifacts of the encoding method we chose and may be omitted by alternative encodings.

x_2	x_1	x_0	y_2	y_1	y_0
0	0	0	1	0	1
0	0	1	0	0	0
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	0	0	1
1	0	1	1	1	1
1	1	0	0	1	1
1	1	1	1	1	0

Table 2 The truth table of the ToySubBytes function

3.2.2 Parameters

The parameters passed to the TASCA instance are the values of the known input plaintext and ciphertexts, as well as the values of the side-channel measurements.

The plaintext and ciphertext values are supplied by fixing the values of S_0 and S_{41}, respectively. In our testing we found that our solver's performance was improved when we combined multiple parameter assignments into a single pseudo-Boolean clause, rather than using one clause per parameter. This is probably because the solver's constraint solving module is more highly optimized than its parsing module.

The side-channel measurement values $M_{t,i}$ are provided as the right-hand side of the pseudo-Boolean measurement equations. Since we model the side-channel data as the Hamming weight of an 8-bit register, these measurement values are always an integer between 0 and 8. The integer parameter $M_{t,i}$ corresponds to the measured Hamming weight of state byte i during subround t, where $t \in [0, 41], i \in [0, 15]$. Again we note that these side-channel measurements exist not only for the state bytes, but also for the key bytes, the round key bytes, and for several additional intermediate values.

3.2.3 State Transition Equations

The AES-128 encryption operation iteratively applies four standard building blocks, or subrounds, to transform the cipher state from plaintext (S_0) into ciphertext (S_{41}). During several of these subrounds the cipher state is also mixed with the expanded cryptographic key. The key expansion process, which converts the initial 128-bit key into 10 128-bit round keys, is also performed using similar building blocks.

The state transition equations we used for each of the four subround types is described below.

– **AddKey/AddRoundKey:** This subround is a straightforward exclusive or (XOR) between the cipher state and the appropriate key or round key. There are several ways of expressing XOR in pseudo-Boolean notation, since it combines binary variables and integer arithmetic. In our testing we

found that the clause that provided the best performance was $a+b-2ab-out=0$, where a and b are binary variables, and out is the binary variable representing the XOR output.

– **SubBytes**: The **SubBytes** subround independently applies an 8-bit to 8-bit substitution operation to each of the 16 state bytes. The SubBytes() transformation is a non-linear byte substitution which cannot be described succinctly as an equation set. We explored several methods of expressing this transformation, which we demonstrate using a toy example ToySubBytes, whose truth table is given in Table 2. First, we attempted to describe each of the 8 output bits as a function of all input bits in sum-of-products form. In the ToySubBytes function, this would result in the following 3 statements:

$$y_2 = \overline{x}_2\overline{x}_1\overline{x}_0 + \overline{x}_2x_1x_0 + x_2\overline{x}_1x_0 + x_2x_1x_0$$

$$y_1 = \overline{x}_2x_1\overline{x}_0 + x_2\overline{x}_1x_0 + x_2x_1\overline{x}_0 + x_2x_1x_0$$

$$y_0 = \overline{x}_2\overline{x}_1\overline{x}_0 + x_2\overline{x}_1\overline{x}_0 + x_2x_1\overline{x}_0 + x_2\overline{x}_1x_0$$

In the case of the real SubBytes function, this construction resulted in 8 very large equations, one for each output bit, each consisting of a sum of 128 8-bit products. Due to the way the SubBytes function was designed, it was not possible to shrink these functions by any meaningful quantity by applying Boolean optimization techniques. Next, we attempted to encode the function in truth-table form, canonically mapping each possible output byte to a single combination of the input bits. In the ToySubBytes function, this would result in the following statement set:

$$\overline{x}_2\overline{x}_1\overline{x}_0 - y_2\overline{y}_1y_0 = 0$$

$$\overline{x}_2\overline{x}_1x_0 - \overline{y}_2\overline{y}_1\overline{y}_0 = 0$$

$$\overline{x}_2x_1\overline{x}_0 - \overline{y}_2y_1\overline{y}_0 = 0$$

$$(etc.)$$

This encoding resulted in 256 short equations, one for each possible output value. Finally, we attempted to write formulas describing the circuit of the most hardware-efficient implementation of the SubBytes function, which we implemented in combinational logic based on the description given in [6]. This encoding was the smallest, resulting in 148 short equations, but it generated several additional variables representing intermediate steps of the computations. In our tests we discovered that the first method resulted in the quickest runtime, even though it created the largest instance files.

– **ShiftRows**: in this subround the state bytes are cyclically shifted within each state row. The actual values of the state bytes are not modified during this state, making it essentially a logical index shuffling operation. Since the state bytes are not modified, this subround does not generate additional side-channel data for our attack. To minimize the size of our AES instances we chose not to explicitly write down equations for this subround. Instead, we folded it into the following MixColumns subround – whenever

a certain state byte was required during the MixColumns subround, we instead referenced the state byte which was to be transformed into this state byte during the ShiftRows operation. For example, instead of writing:

$$[S_{3,0}, S_{3,1}, S_{3,2}, S_{3,3}] = [S_{2,0}, S_{2,5}, S_{2,10}, S_{2,15}]$$

$$[S_{4,0}, S_{4,1}, S_{4,2}, S_{4,3}] = ColumnTransform\left([S_{3,0}, S_{3,1}, S_{3,2}, S_{3,3}]\right)$$

We wrote:

$$[S_{4,0}, S_{4,1}, S_{4,2}, S_{4,3}] = ColumnTransform\left([S_{2,0}, S_{2,5}, S_{2,10}, S_{2,15}]\right)$$

– **MixColumns**: This subround applies a linear transformation to the state one column at a time, where each column consists of 4 state bytes. This is a 32-bit operation which is repeated 4 times, once for every column in the state. MixColumns has an efficient implementation using 8-bit words, described in detail in [8, §5.1 and §2.1.3], which mostly consists of bitwise shifts and exclusive or operations. Since our DUT is an 8-bit microcontroller, it uses this efficient implementation. As a result, the intermediate steps of the 8-bit algorithm generate additional side-channel leaks, and as such are also relevant to us as attackers.

3.2.4 Key Expansion Equations

This set of equations implements an invertible key derivation function mapping between the secret key variables $K_{i,j}$ and the round key variables $RK_{r,i,j}$. As specified in the AES standard [8, §4.3.1], the round keys for the various rounds are generated by applying a series of XORs with previous round key values and with hard-coded values called round constants, and by invocation of the standard SubBytes operation. For example, the mapping between the secret key $K$and the first round key RK_0, using a helper variable $RK_{0,a}$, is described by the following equation set:

$$RK_{0,a,0} = SubBytes\left(K_{13}\right)$$

$$RK_{0,a,1} = SubBytes\left(K_{14}\right)$$

$$RK_{0,a,2} = SubBytes\left(K_{15}\right)$$

$$RK_{0,a,3} = SubBytes\left(K_{12}\right)$$

$$RK_{0,0} = RK_{0,a,0} \oplus K_0 \oplus 0\text{x}01$$

$$RK_{0,1} = RK_{0,a,1} \oplus K_1$$

$$RK_{0,2} = RK_{0,a,2} \oplus K_2$$

$$RK_{0,3} = RK_{0,a,3} \oplus K_3$$

$$RK_{0,4} = RK_{0,0} \oplus K_4$$

$$RK_{0,5} = RK_{0,1} \oplus K_5$$

$$RK_{0,6} = RK_{0,2} \oplus K_6$$

$$RK_{0,7} = RK_{0,3} \oplus K_7$$

$$RK_{0,8} = RK_{0,4} \oplus K_8$$

$$RK_{0,9} = RK_{0,5} \oplus K_9$$

$$RK_{0,10} = RK_{0,6} \oplus K_{10}$$

$$RK_{0,11} = RK_{0,7} \oplus K_{11}$$

$$RK_{0,12} = RK_{0,8} \oplus K_{12}$$

$$RK_{0,13} = RK_{0,9} \oplus K_{13}$$

$$RK_{0,14} = RK_{0,10} \oplus K_{14}$$

$$RK_{0,15} = RK_{0,11} \oplus K_{15}$$

We applied the same considerations listed in the previous Subsection to our choice of optimal representation for the XOR and SubBytes operations.

In the most common usage of AES, the key bytes are expanded as soon as the key is loaded to the cryptographic device. The expanded round key bytes are then used for multiple encryption operations, and only the encryption operations (but not the initial key expansion) are monitored by the attacker. It is thus commonly assumed that the attacker has side-channel information only for the encryption operation, but not for the key expansion operation.

3.2.5 Side-Channel Equations

The device under test has a power consumption proportional to the Hamming weight of the 8-bit word it is currently processing. Since these measurements are generated whenever the DUT processes a byte, there exists such a measurement equation for each byte of the state after each subround. There are also measurement equations for the individual bytes of the key and of the round keys, which are processed by the DUT during the **AddKey/AddRoundKey** operations. Finally, there are multiple measurement equations for intermediate values generated by the 8-bit microprocessor as it performs the 32-bit **MixColumns** subround, as described in the previous Subsection. As we discuss further in Section 4, it is sometimes beneficial to include fewer measurement equations, and by doing so limit the effect of measurement errors on the equation set.

3.2.6 The Objective Function

In an algebraic side-channel attack, the attackers objective can be succinctly described as follows:

> Given the algorithmic description of a cryptographic algorithm, the physical power model of the device under attack and the side-channel measurements, output a key assignment for which the expected side-channel information is as close as possible to the measured side-channel information.

To create the objective function, we recast this problem as an **optimization problem:**

> Find the **minimal** assignment to an **error vector** such that it is possible for the **cryptographic algorithm**, operating under a certain unknown **key** and in a certain **physical power model,** to produce the **measured side-channel information** affected by this error**.**

Our objective is thus to minimize the use of error variables to modify the measurement equations. By writing the objective function as a simple sum of these error variables we achieve a basic, yet effective, method of guiding the solver toward the correct solution. We show in [23] how a more elaborate objective function can use additional outputs from the decoding process, such as decoder confidence information, to guide the solver into trying to use certain error variables before others.

3.2.7 Summary

Following the notation described in the previous Subsections, the constraint problem corresponding to a TASCA instance covering subrounds 1 to 5 can be written as follows:

$$Min : \sum e^{+}_{t,i} + \sum e^{-}_{t,i}, \text{ s.t.:}$$

$$S_0 = Plaintext$$

$$RK_0 = KeyExpansion\,(K)$$

$$S_1 = S_0 \oplus K$$

$$S_2 = SubBytes(S_1)$$

$$S_4 = MixColumns(ShiftRows(S_2))$$

$$S_5 = S_4 \oplus RK_0$$

$$\sum_{j=0}^{7} S_{t,i,j} + e^{+}_{t,i} - e^{-}_{t,i} = M_{t,i}$$

Note that the cipher state at the end of the ShiftRows operation (S_3) is not written explicitly. Also note that the assignment of the ciphertext to S_{41} is omitted, as the equation set only includes the first 5 subrounds and as such any assignment to the ciphertext will be simply optimized away.

A full TASCA instance constructed using this methodology consists of approximately 6,500 variables and 6,000 constraints, which are reduced to approximately 5,000 variables and 5,000 constraints during the SCIP solver's presolving phase. The instance occupies approximately 1.3 megabytes of disk space in OPB format. A representative constraint problem with this structure can be found in the Appendix, and a set of full-length instances can be found in the contributed data set accompanying this article, as further described in Subsection 8.3.

3.3 The TASCA Problem Space

The TASCA solver is presented with **leak equations** and with **auxiliary information** such as known plaintext and ciphertext, and is tasked with finding the secret key.

At the most extreme case of limited information, the solver is provided only with equations describing the encryption algorithm, but with no leak equations at all and with no auxiliary information. It is immediately evident that this instance is **trivial to solve** but will return an **incorrect key**, since any key will satisfy this extremely underspecified problem. The same holds if no leaks are known and only the plaintext or the ciphertext (but not both) are provided as auxiliary information.

In contrast, if the plaintext and ciphertext are *both* provided, and no leak equations exist, the problem degrades to that of standard algebraic cryptanalysis – with high probability only a single key exists that will satisfy this plaintext-ciphertext relation, but this cryptanalytic instance is **too difficult to solve**, making it impossible for a solver to find the key in a reasonable time[16].

Another extreme case is the case where all side-channel data is presented to the solver, without any measurement error. As shown in [27], even with no auxiliary information the solver has enough information to **solve the problem successfully and efficiently** and find the correct key in a reasonable time. Obviously, any additional auxiliary information such as known plaintext or ciphertext only increases the probability of a successful outcome.

The amount of leak information provided to the solver is measured both by the number of measurement equations and by the amount of error tolerance built into each equation. As opposed to ASCA problems, for which the leak equations are precisely defined, TASCA (and Set-ASCA) leak equations admit several possible values for each leak, allowing the solver to tolerate some amount of noise in the measurements.

Incorrect Key Bytes	0	1	2	3	4	5
AES operations required	1	2^{12}	2^{24}	2^{36}	2^{48}	2^{60}
Estimated time using AES-NI[3]	<1 sec	<1 sec	<1 sec	25 sec	24 hr	9 yr

Table 3 Estimated time for brute-force searching for incorrect key candidates

3.4 Dealing with Solver Failures

The solver does not always succeed in recovering the correct key. There are three possible failure modes:

- The first type of failure occurs when the solver reports that the problem is **unsatisfiable**. The source of this failure is a **decoding failure** – the decoder introduces more noise into the leak equations than the solver is capable of dealing with.
- The second type of failure occurs when the solver **times out** and does not return any answer within the specified time.
- The third type of failure occurs when the solver **returns an incorrect key**. Typically this failure is caused by too few leak equations, or too much error tolerance, which causes the problem to be under-defined.

A specific type of the third failure mode, which we discovered to be quite common, results in the solver returning a **partially correct key**. Due to the specific byte-oriented micro-architecture of AES, this mode is usually characterized by a partition of the key bytes into a group of perfectly correct bytes and a group of wrong bytes. Since the error tends to be localized to only a few bytes, the key can be recovered in some cases from the incorrect result using a moderate amount of brute-force searching (since we have a plaintext/ciphertext pair).

To analyze the brute-force effort required by an attacker, assume that e of the 16 AES key-bytes are incorrect, and that the rest are correct. The attacker must go over all $\binom{16}{e} \approx 2^{4e}$ possible locations for those errored bytes, then try $256^e = 2^{8e}$ possible candidate assignments for these positions, resulting in an approximate total effort of $2^{4e} \cdot 2^{8e} = 2^{12e}$ AES operations. Most modern Intel CPUs have a native implementation of AES (AES-NI), which allows a sustained rate of more than 2^{31} AES operations per second on a contemporary system[3]. As illustrated below in Table 3, an attacker can use a single machine with an AES-NI implementation to probe the close neighborhood of a candidate key and find the correct key within several hours, even if as many as 4 of the 16 bytes returned are incorrect.

4 Theoretical Modeling of Error Tolerance

4.1 Measuring Error Tolerance

Since there are different ways of encoding error into an equation set, it is difficult to compare the ability of different methods to deal with errors. Therefore,

it is useful to find a way to discuss errors in terms of standard metrics such as signal to noise ratio or bit error rate.

As stated in Section 3, the side-channel leak is first passed through a decoder, where it is converted to set of leak equations, and next to a solver. The equation representations we discuss here deal with errors by permitting more than one valid assignment to each individual leak (i.e. they follow either the Set-ASCA or the TASCA methodologies). Thus, a set of k values are accepted for each leak. A necessary condition for the solver (Set-ASCA or TASCA) to succeed is that the correct value of each leak is a member of the acceptable set presented to the solver[2].

Let us now assume that we wish to carry out an attack based on Hamming weights of internal calculations performed by an 8-bit micro-controller. In this model each leak x_i is an integer value which takes a value between 0 and 8. The leak is modulated onto the amplitude of the power trace, subjected to some additive noise, and finally recovered by the attacker.

As shown in [14], in the case of an 8-bit micro-controller the amplitude of the power trace is approximately linearly dependent on the Hamming weight. To recover the Hamming weight from the power trace, the attacker typically measures the amplitude of the power trace at one or more points in time, then applies an *affine transform* to these measurements to arrive at $\hat{x}_i$, the estimated value for the leak x_i. This estimated value can thus be seen as the result of the original integral leak x_i and an additive Gaussian noise element $\nu_i \sim \mathcal{N}(0, \sigma)$: $\hat{x_i} = x_i + \nu_i$.

To put this model into standard engineering signal and noise terms, the signal power P_s can be defined as the *variance of the Hamming weights of the inputs*, assuming a uniform distribution of the inputs. The noise power P_n can be defined as the *variance of the noise* ν. This leads to the standard definition of signal-to-noise ratio:

Definition 1 Let $HW(i)$ denote the Hamming weight of an integer $i \in [0, 255]$

Proposition 1 $SNR \approx -20 \log_{10} \sigma + 3$

Proof

$$SNR = 10 \log_{10} \frac{P_s}{P_n}$$

$$= 10 \log_{10} \frac{\frac{1}{256} \sum_{i=0}^{255} \left(HW(i) - E\left[HW(i)\right]\right)^2}{\sigma^2}$$

Since $E\left[HW(i)\right] = 4$ it can be verified that the numerator equals 2, giving

$$SNR = 10 \log_{10} 2 - 10 \log_{10} \sigma^2$$

$$\approx -20 \log_{10} \sigma + 3$$

[2] This is not a sufficient condition – even if all leaks are recovered correctly the problem may still be under-defined or computationally intractable.

Now that $\hat{x}_i$ is defined, it needs to be presented to the solver. Assuming the solver is a TASCA or Set-ASCA solver with a set of size k, the natural approach would be for the decoder to populate the set of valid solutions with the k integer values closest to $\hat{x}$. A necessary condition for the solver to succeed would then be that one of the elements of the set output by the decoder is the original x_i. The elements of this set may all be considered equally desirable (in the case of Set-ASCA), or otherwise ranked according to their distance from $\hat{x}_i$ (in the case of TASCA). For example, if the set size k is 1 (corresponding to ASCA) and the decoder observation was $\hat{x}_i = 2.2$, the set will contain the value 2. In other words, if the decoder outputs the symbol closest to $\hat{x_i}$, and the original symbol x_i was 2, the attack will succeed only if the decoded value $\hat{x_i}$ is between 1.5 and 2.5, or equivalently if the noise ν_i is between -0.5 and 0.5. If k is 2, the equations will allow the values 2 and 3, and the attack will succeed only if $\hat{x} \in (1.5, 3.5)$. In general, if k values are acceptable then the noise ν_i must be in the range $\left[-\frac{k}{2}, \frac{k}{2}\right]$. For a **decoding success** we require this condition to hold for all m leaks in the leak vector simultaneously. Assuming the noise is i.i.d. for all leaks, the probability of such an event is:

$$
\begin{aligned}
&\Pr\left(\text{Decoding Success}\right) \\
&= P\left(\hat{x}_i \in \left[x_i - \frac{k}{2}, x_i + \frac{k}{2}\right] | x_i\right)^m \\
&= P\left(x_i + \nu_i \in \left[x_i - \frac{k}{2}, x_i + \frac{k}{2}\right] | x_i\right)^m \\
&= P\left(\nu_i \in \left[-\frac{k}{2}, \frac{k}{2}\right] | x_i\right)^m \\
&= P\left(\nu_i \in \left[-\frac{k}{2}, \frac{k}{2}\right]\right)^m \\
&= \left(\frac{1}{\sqrt{2\pi\sigma^2}} \int_{-\frac{k}{2}}^{\frac{k}{2}} e^{-\frac{x^2}{2\sigma^2}} dx\right)^m
\end{aligned}
$$

By fixing the set size k and the number of leaks m and solving for σ, we can apply Proposition 1 and find the maximum Gaussian noise power tolerable by sets of size k if a certain success probability is desired.

If we define the per-leak error rate as the probability that a set of size 1 does not contain the correct value of x_i, we can convert the value of σ to an error rate using the following relation:

Definition 2

$$
P_{error} = P\left(\nu_i \notin \left[-\frac{1}{2}, \frac{1}{2}\right]\right) = 1 - \frac{1}{\sqrt{2\pi\sigma^2}} \int_{-\frac{1}{2}}^{\frac{1}{2}} e^{-\frac{x^2}{2\sigma^2}} dx
$$

Leak Count	m =100 leaks		m =200 leaks	
Set size k	min. SNR [dB]	max. error rate	min. SNR [dB]	max. error rate
1	20.8	0.1%	21.2	0.05%
2	14.8	5.2%	15.2	4.3%
3	11.3	19.5%	11.6	17.7%
4	8.8	33.1%	9.1	31.1%

Table 4 Relation between error rate and set size for normally distributed noise when we require Pr (Decoding Success) = 99%

Table 4 summarizes these results for relevant ranges of parameters. It can be seen from the table that each set size corresponds to a certain maximal error rate and, equivalently, to a certain minimal signal-to-noise ratio. We can see that for a set size $k = 1$ (corresponding to the ASCA method) the maximum error rate is between 0.05% and 0.1% for $m \in [100, 200]$. Such a low error rate is impractical to achieve even with high-end measurement equipment. However, with $k = 3$ it is possible to tolerate an error rate of almost 20%, which is within reason.

Figure 2 displays the probability that a decoder using a set of size $k = 3$ will correctly capture all measurements with a fixed error rate. It can be seen that as the number of equations m grows, the probability of decoding success falls exponentially. Therefore, we see that an increase in information (represented by a higher number of leak equations) causes a decrease in the robustness of the attack, making it more sensitive to errors. Thus (and perhaps counter-intuitively) it makes sense to provide the solver with the *least* amount of side-channel information required for a successful key recovery, even if more information is available.

4.2 Evaluating Solver Performance

A central component of this attack is a pseudo-Boolean optimizer, described in more detail in [2]. There are several ways of constructing such a solver, for example by extending a SAT solver or by constraining an Integer Programming solver.

When evaluating the performance of various solving methodologies, we used the following metrics of performance:

1. The **time** it takes the solver before the key is successfully recovered. This metric gives an advantage to SAT solvers, which are simpler in construction than PB optimizers and are dramatically faster for instances of similar size.
2. The **success probability** of the attack to succeed, given a typical power trace. While this is an important practical consideration, we note that from a security standpoint an attack that can break AES for a small but significant fraction of the instances it encounters is arguably just as dangerous as one that can break nearly all instances.
3. The **amount of leaks** required – naturally, the attack which requires the least amount of leaks is the most useful. We consider this metric signifi-

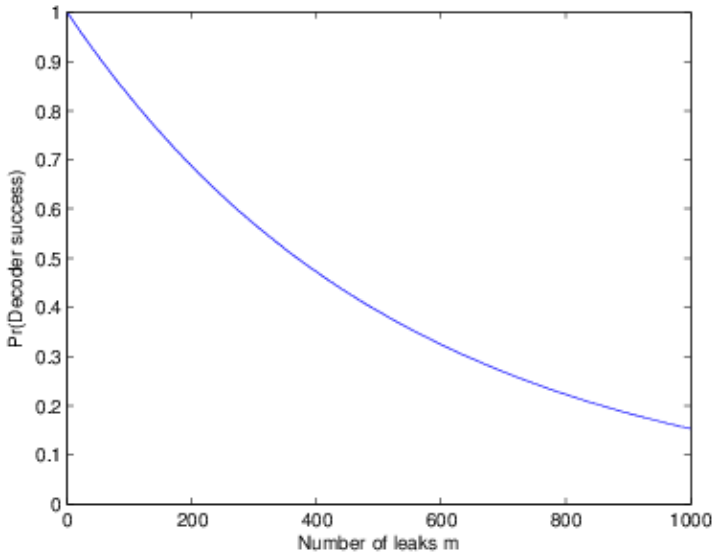


Fig. 2 Decoder success rate as a function of leak count m, with a fixed set size $k = 3$ and an error rate of 30% ($\sigma = 0.4824$, SNR=9.3dB)

cant, especially considering the large amount of profiling and preprocessing involved in a complete attack and the interplay between the amount of required leaks and the maximum tolerable error rate (see Section 4).

4. The ability of the solver to **tolerate noise** and error. There are several conflicting ways of treating this metric and we will attempt to unify them in this discussion.

5 Experiment Setup

In this section we describe the setup we used for our experiments with ASCA, Set-ASCA, and TASCA.

5.1 The Device Under Test

As a device under test we chose an 8-bit micro-controller implementation of AES-128, based on the standard NIST implementation of [20] and on a hardware implementation specified in [29]. We assumed that the device leaks the Hamming weight of the 8-bit operand on its data bus. Following the observations of Section 4, we aimed to provide the minimum amount of (errored) measurements to the solver. Thus, we only modeled the first few rounds of AES. We made the following assumptions:

– Only the **plaintext** is provided to the solver, but not the ciphertext.

- The leaks from the **key expansion process** are not available to the solver, since we assume that the DUT performs round key expansion in advance.[3]
- The **AddKey** and **AddRoundKey** operations leak the Hamming weights of the 16 state bytes after the XOR with the key/round key, as well as the Hamming weights of the key bytes themselves, giving a total of 32 leaks per subround.
- The **SubBytes** operation is implemented as a look-up table (LUT). The equations representing the SubBytes operation use the canonical representation (see [22, §5.2]), using a single equation per output bit or a total of 8 equations per invocation of SubBytes. The LUT operation leaks only the Hamming weights of the 16 state bytes after the SubBytes operation (and not any other internal state information), for a total of 16 leaks per subround.
- The **ShiftRows** operation is implemented logically as index shuffling and as such does not leak any information.
- The **MixColumns** operation is implemented using 8-bit XTIME and XOR operations as specified in [20] 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 (consisting of **SubBytes, ShiftRows, MixColumns** and **AddKey/AddRoundKey**) leaks 100 Hamming weights of 8-bit values.

The experiments were performed under various error rates and set sizes up to the theoretical thresholds calculated in Section 4. We chose to focus on the performance of the solver and not on that of the decoder, assuming that the leaks satisfied the necessary condition for **decoding success** outlined in Section 4. To apply an error rate of e to the Hamming weight measurement vector of length m for a given experiment, em indices were chosen independently at random, and the Hamming weight at each of these selected indices was modified by either ± 1 (for sets of size $k = 3$) or by $+1$ (for sets of size $k = 2$). Sample AES instances created using this method are available online [26] and are described in the Appendix.

Each ASCA or TASCA instance consists of a general description of the cryptographic algorithm as a set of equations, an assignment of any known inputs to the algorithm, a specification of the measurement setup, and finally a set of potentially errored measurements. For TASCA instances we also include a goal function, which instructs the solver to aim for a solution (i.e. key assignment) which minimizes the amount of noise in the measurements. If we change the measurement equations so that they do not admit noise (i.e., let $k = 1$) and eliminate the goal function, we transform the problem from a TASCA instance to an ASCA instance similar in form to that used in [28].

[3] It was already established in [13] that the Hamming weights leaked from an 8-bit microcontroller implementation of AES during key expansion are sufficient for full key recovery, even without any additional state information.

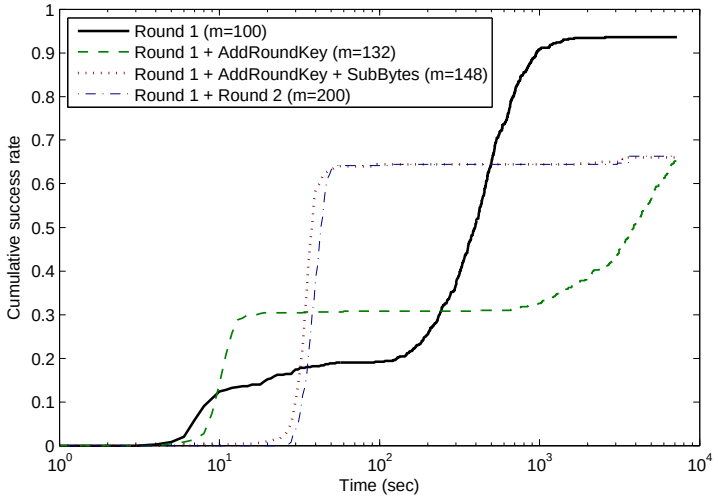


Fig. 3 ASCA cumulative success rates for different amounts of side-channel data

5.2 Solver Software and Hardware

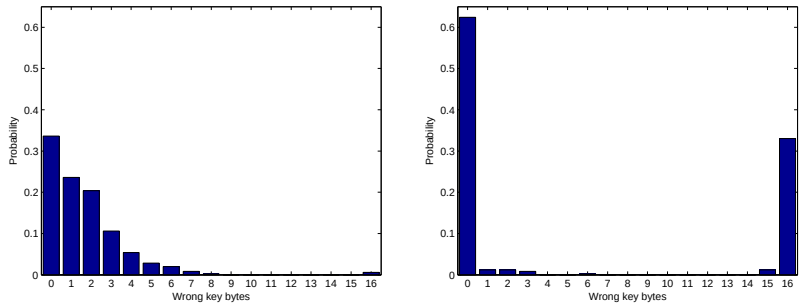
The solver used in our experiments was SCIP version 1.2.0 compiled for Windows 64-bit[4]. SCIP is a general purpose MIP solver with robust support for pseudo-Boolean optimization instances. When we began running our experiments in late 2009, this solver was listed by [15] as the best non-commercial solver available for non-linear optimization problems. 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. It should be noted that running a single instance at a time will result in noticeably better performance due to less contention on the L2 cache (equation solving is very RAM intensive) and due to Intel's Turbo Boost feature which speeds up one computational core when the others are idle [10]. The running time of each instance was limited by a shell script – ASCA and Set-ASCA instances were limited to two hours and TASCA instances limited to two days. A set of MATLAB scripts was used to create random instances, run the solver and collect the results automatically.

6 Results – ASCA with No Errors

An ASCA instance typically has a smaller solution space than a TASCA instance, and is thus much easier to solve. We used ASCA instances to determine the minimal amount of side-channel information which can be provided to the solver for an efficient and correct response.

Amount of side-channel data	Mean solving time	Success rate after...		
		10 sec	80 sec	2 hrs
Round 1 (100 leaks)	392 sec	12.4%	19%	93.6%
Round 1 + AddRoundKey(132 leaks)	1950 sec	14.2%	30.8%	65.6%
Round 1 + AddRoundKey + SubBytes (148 leaks)	102 sec	0%	64%	66%
Round 1 + Round 2 (200 leaks)	137 sec	0%	64.2%	66.3%

Table 5 Success rate of the ASCA solver at different times



(a) Round 1 (100 side-channel measurements)

(b) Round 1 + AddRoundKey (132 side-channel measurements)

Fig. 4 ASCA wrong key histograms

Figure 3 shows the success probability of non-error-tolerant ASCA measurements as a function of the run-time with different amounts of side-channel data. It can be seen that 20%-60% of the problems are solved within about 1 minute while the rest run for a significantly longer amount of time. This agrees with the findings of [27].

Table 5 summarizes the success rate of the ASCA solver as a function of time for different amounts of side-channel data. Based on Section 3.4, we consider as success cases where the solver terminates and returns a key which is at most 4 bytes apart from the correct one .

Based on these results we conclude that very few error-free side-channel measurements – a single AES round is sufficient – can usually achieve full key recovery. We discovered that if we reduced the leak count to less than a full round, the probability of success became vanishingly small. Note that the success probability after 2 hours is largely unaffected by the amount of side-channel information. Thus it makes sense to use only a single round of leaks and no more, and thus increase the robustness of the attack.

A possible drawback to using a small amount of side-channel data is the risk of a producing an underspecified problem, causing the solver to return an incorrect answer. To demonstrate the impact of this effect, Figure 4 shows the distribution of incorrect key bytes in the recovered solution for 100 and 132 of side-channel measurements, where instances which did not terminate after 2 hours were assigned the all-zero key.

We can see that with 100 leaks, only 6.4% of the recovered keys had more than 4 incorrect key bytes. With 132 measurements this rate drops to 0.2%, but an additional 35% of the instances fail with a solver time-out. With additional measurements the fraction of wrong bytes drops to nearly 0 (graphs omitted). As discussed previously in Subsection 3.4, even 3 or 4 incorrect key bytes can be considered a correct result, if we allow the immediate neighborhood of the candidate key to be probed using brute force. Under this assumption, in all cases the key was either recovered correctly or the operation timed out. This leads us to conclude that 100 leaks (one full AES round) are enough to uniquely determine the key in the case of error-free ASCA (i.e., $k = 1$), even if the ciphertext is not provided to the solver.

7 Results – TASCA and Set-ASCA with Errors

This section describes the results of TASCA and Set-ASCA runs on simulated power traces of AES which have been corrupted with noise. We tested the average solving time and average success rates for various combinations of side-channel information amounts and error rates. We also measured the effect of the optimizing aspect of the solver (the goal function) on the performance of our solver.

7.1 Effect of Error Rate

The objective of this experiment was to measure the effect of the error rate on the solving time and success probability of our TASCA solver. Based on the conclusions of the previous section, we provided our TASCA solver with one round of side-channel leaks (m =100 measurements) and with a known plaintext. The error rate was chosen to be between 0% and 25%. According to Table 4, a decoder has at least a 99% probability of success when producing such a set of equations, given the above parameters and a set of size k =3. Figure 5 shows our results. We also ran the experiment with $k = 2$ and error rates of up to 5% (results omitted).

In general, the TASCA approach showed itself capable of recovering the secret key, with a success probability of 30%-80%, from errored traces in 6 to 24 hours for sets of size $k = 2$ and $k = 3$, even with error rates all the way up to the theoretical boundary of 19.5% previously calculated in Subsection 4. Interestingly, we could find no significant correlation between the error rate and the success rate, nor between the error rate and the solving time. This is in contrast to our previous results on Keeloq [22], where increasing the error rate caused a measurable increase in the running time and a corresponding decrease in the success rate.

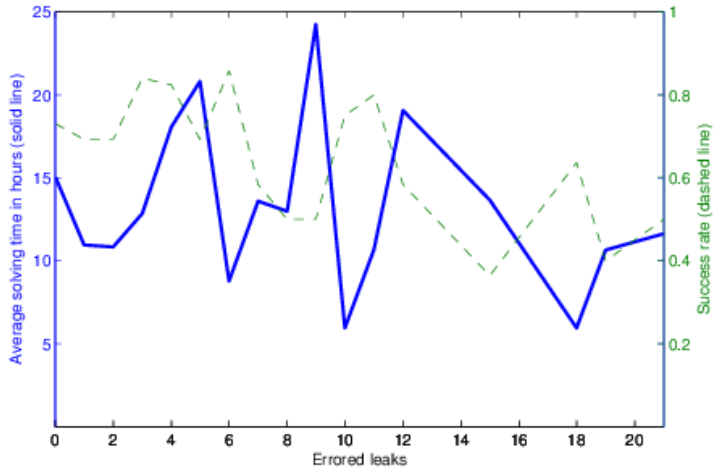


Fig. 5 Effect of error rate on success rate and solving time for 3-set TASCA of AES with $m = 100$ leaks (20 experiments per data point)

Set Size k		1	2	3	4
Success probability	(TASCA)	78%	87.5%	73%	72.7%
	(Set-ASCA)	78%	9.2%	0%	0%
Mean solving time in minutes	(TASCA)	6.45	245.2	901.99	1332.07
	(Set-ASCA)	6.45	171.05	7.18	2.92

Table 6 TASCA vs. Set-ASCA performance with 100 leaks

7.2 Comparing TASCA and Set-ASCA

As discussed in Subsection 4.2, an alternative approach to TASCA called Set-ASCA was introduced in [27] and more recently investigated in [31]. In this approach the k values in the set are each equally acceptable, that is, there is no incentive for the solver to choose one value over another. The Set-ASCA equation set, which is essentially similar to the straight ASCA equation set, is then submitted to a standard SAT solver. Assuming an identical quantity of leaks is used in both cases, Set-ASCA has been shown in [31] to be about 20 times faster than (optimizing) TASCA. However, many keys – potentially an exponential amount – may satisfy the same side-channel leak equations. If a non-optimizing solver is used in this case, it will arbitrarily choose a satisfying solution and terminate, making its success probability exponentially small. In contrast, an optimizing solver will only terminate once it has found a solution which minimizes the goal function. Assuming the goal function has been correctly specified, we hypothesized that an optimizing TASCA approach is more likely to find the correct key than a random satisfying assignment.

To investigate this scenario, we compare a TASCA and a Set-ASCA solver operating on m =100 leaks of side-channel data with 0% errors, varying the set size from $k = 1$ (standard ASCA) to $k = 4$. Table 6 describes our results.

Note that for a set size of k =1, both TASCA and Set-ASCA solvers are reduced to the case of standard ASCA, making their performance identical. As we expected, the solving time of the optimizer is much worse than that of the SAT solver, a difference that only grows as the set size grows. On the other hand, it can clearly be seen that the accuracy of Set-ASCA falls dramatically when we must increase the set size k to overcome decoder failures: the SAT solver used in Set-ASCA may terminate quickly, but it provides the correct key in only 9% of the cases for $k = 2$ and is virtually always wrong for $k = 3$ or $k = 4$. This is caused by the large set of admissible solutions made possible by the lower precision of the leak equations. Since the SAT solver chooses one of these solutions arbitrarily, its success rate falls dramatically as the set size grows. In contrast, the optimizer is motivated to choose the best solution from the available set. As a result, its success rate is largely determined by the error rate (as we saw in Subsection 7.1) and not by the set size.

8 Conclusions and Discussion

8.1 Comparison with Keeloq Results

This paper describes an attack on the block cipher AES [20]. A previous work [22] applied TASCA to Keeloq ‘[9], a simple stream cipher used in car remote controls and other low-security applications. There are several differences between the Keeloq cipher and the AES cipher, when viewed from the perspective of a TASCA attack. On one hand, the Keeloq cipher seems easier to analyze because of its weak diffusion property – the key bits are shifted into the Keeloq cipher state one bit at a time, whereas in 8-bit AES they are introduced into the cipher state one byte at a time. On the other hand, the Keeloq implementation runs on an ASIC circuit which leaks 32-bit Hamming distances, a side-channel leak which is generally considered more difficult to attack in contrast to the 8-bit Hamming weights leaked by the AES implementation.

Table 7 contrasts AES and Keeloq instances in terms of their respective sizes and solving times. The Keeloq instances use m =90 subrounds (and hence 90 measurements), while the AES instances use one AES round with m =100 measurements. In both cases the TASCA instances use a set size of $k = 3$ and the leaks are corrupted with a 5% error rate. It can be seen from the table that despite the fact that the AES instance is only 9 times larger than the Keeloq instance, it is harder to solve by three orders of magnitude. Another interesting difference, which we demonstrated in Figure 5, is the low correlation between the error rate and the solving time for AES – as shown in [22], the solving time of Keeloq TASCA instances increases super-linearly with the error rate, while we could observe no such condition in the case of AES. In our experiments, AES instances took roughly the same amount of time to solve regardless of their error rate.

Both differences in behaviour between AES and Keeloq may be attributed to the different diffusion properties of the two ciphers. While in Keeloq the

	Keeloq	AES	Ratio
ASCA instance size	140KB	1.3MB	x9.3
ASCA instance time	0.36 sec	387 sec	x1072
TASCA instance size	140KB	1.3MB	x9.3
TASCA instance time	22 sec	8.7hr	x1436

Table 7 Comparison of AES and Keeloq performance

key is introduced into the equation one bit at a time, in AES it is XORed into the plaintext as soon as encryption begins and further diffused by the following operations. The good diffusion property of AES makes it difficult for the solver to infer the value of one variable from the assignment of another. This both increases the run-time and defeats the "added value" granted to the solver for correctly determining a partial solution. Since diffusion has such a profound effect, finding a more precise power model of the cipher which allows additional leaks to be considered and reduces this diffusion property should make AES instances easier to solve.

8.2 A Strategy for Successful TASCA Attacks

As shown in this report, the choice of operating parameters can have a substantial effect on the running time and success probability of an algebraic side-channel attack. Increasing the amount of side-channel measurements available to the solver has a mixed effect: On one hand, as shown in [31], it decreases the space of possible solutions sufficiently to allow the use of non-optimizing solvers with better running times. On the other hand, it increases the sensitivity of the solver to noise. For a fixed amount of side-channel leaks, we showed that increasing the size of the set of acceptable values per leak (k) increases the running time but does not decrease the success probability of the attack.

In view of these findings, we can recommend that implementers first characterise the expected signal-to-noise ratio of the DUT using standard signal processing techniques; Next, they should choose the minimal amount of side-channel leaks (m) required for a successful key recovery; Finally, they should choose the minimal set size (k) which can tolerate the expected amount of noise with good probability. Specifically in the case of AES, we discovered that using m =100 leaks is a good choice for signal to noise ratios of as low as 20dB for k =1, up to 14dB for k =2, and up to 11dB for k =3.

8.3 Contributed Data

The TASCA instances we created during our research may be interesting to developers of constraint programming tools. Our instances have a highly regular structure due to the structure of symmetric block ciphers. Our experiments indicate that the solving difficulty of these instances is largely independent of the variable/clause ratio, but instead depends on domain-specific properties

such as the signal-to-noise ratio of the power trace. We have made available data sets both for the Keeloq and the AES ciphers, and both for the Hamming weight leakage model and for more elaborate leakages. These data sets are all available on our website [25,26]. We have also submitted some of our instances to the yearly pseudo-Boolean competition [15] under the non-linear optimization category. Quite interestingly, the solving performance of these instances has improved considerably between consecutive PB competitions. For example, our most difficult Keeloq instance, *90_rounds_15_errors.opb*, took us 2990 seconds (49 minutes) to solve using SCIP version 1.2.0. In the 2011 pseudo-Boolean competition the same instance was solved in 450 seconds (7.5 minutes) by SCIP version 2.0.1.4. Finally, in 2012 the same instance was solved in only 50 seconds by SCIP 2.1.1.4 – a 60-fold increase in performance.

8.4 Practical Considerations for Solver Authors

As we evaluated different open-source solvers, we found that there are several properties which make some solvers better suited than others for our specific attack. Our observations may be of interest to writers and designers of constraint solving systems:

1. **Robust input handling**: We wrote our equation set in the variant of the OPB format specified by Manquinho and Rousselin in [15]. We chose this format to allow us to easily switch between the different submissions to the PB competition and evaluate their performance. However, we discovered that some solvers crashed or behaved unpredictably when receiving extremely large or improperly formatted input files. In addition, several solvers did not allow us to use meaningful variable names, requiring that all variables be named "x1", "x2", etc. Solver designers should take note that some instances may be hand-written and contain errors and domain-specific variable names, and be prepared to deal with such inputs.
2. **More detailed outputs:** The typical outputs produced by the solvers we evaluated contained either a description of the optimal instance – that is, the satisfying assignment and the corresponding value of the goal function – or an opaque claim of unsatisfiability. We were also interested in more detailed outputs in both cases – for satisfiable instances, we would like the solver to provide descriptions of satisfying but suboptimal assignments it considered and later rejected. In the case of unsatisfiability, we were interested in learning about the minimal clause set required to cause a logical contradiction. Finally, we were interested in any partial assignments discovered by the solver, even if the instance ultimately times out. We would have been able to use these partial outputs as the starting point to a cryptographic brute-force attack which quickly goes over all unassigned variables.
3. **Snapshots and suspend/resume functionality:** Some of our instances took several weeks to run. While these instances were running we had extremely limited information about their progress, mainly based on periodic

outputs to a log file. In addition, any intentional or unintentional restart of the solver hardware (due to e.g. security updates, power failures, etc.) caused us to lose many days of computation time. We would have liked the solver to periodically save its current run-time state to disk for detailed analysis. More importantly, we would have wanted the ability to back up these snapshots, then restart the solver based on such a snapshot.

4. **Indication of key variables:** Some of our AES instances were over 5MB of size and contained more than 20,000 variables and clauses. Many of the variables in the instance described internal technical states of the AES cipher which were of no practical interest to us as attackers. In fact, we were only ultimately interested in the 128 variables describing the secret key bits. Since our solver had a "pre-solving" phase, at which it applied various Boolean optimizations to the problem instance to make it more efficient to solve, we would have liked the ability to indicate these critical variables ahead of time, to ensure they are not "optimized away" during the pre-solving step.

8.5 Conclusion

This report shows how optimizing constraint solvers can be applied to side-channel cryptanalysis. The noise-tolerant TASCA approach, which was previously applied to the low-security Keeloq cipher, was shown to be usable for full-strength ciphers such as AES. The secret key can be recovered from 60%-70% of AES instances even when only a single trace is provided, and even when 20% of the trace signal is corrupted by noise. This new cryptanalytic capability may compromise secure systems whose defense against (statistical) side-channel attacks was an aggressive re-keying schedule which results in a small amount of traces per given key – as we showed, even a single encryption is enough to recover the key, assuming that the device under attack has been properly profiled by the attacker.

Acknowledgements The authors thank the anonymous reviewers for their detailed and helpful suggestions. The authors wish to acknowledge Mario Kirschbaum, Thomas Popp, Mathieu Renauld and Francois-Xavier Standaert for their contributions to this research.

References

1. http://www.msoos.org/cryptominisat2/.
2. T. Achterberg. *Constraint Integer Programming.* PhD thesis, Technische Universität Berlin, 2007.
3. Kahraman Akdemir, Martin Dixon, Wajdi Feghali, Patrick Fay, Vinodh Gopal, Jim Guilford, Erdinc Ozturc, Gil Worlich, and Ronen Zohar. Breakthrough AES Performance with Intel AES New Instructions. Technical report, Intel Corporation, October 2010. `http://software.intel.com/file/27067`.
4. T. Berthold, S. Heinz, M. E. Pfetsch, and M. Winkler. SCIP – solving constraint integer programs. SAT 2009 competitive events booklet, 2009. `http://www.cril.univ-artois.fr/SAT09/solvers/booklet.pdf`.

5. Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In *CHES*, pages 450–466, 2007.
6. D. Canright. A very compact S-Box for AES. In J. R. Rao and B. Sunar, editors, *CHES 2005*, volume 3659 of *LNCS*, pages 441–455. Springer, 2005.
7. Nicolas T. Courtois and Gregory V. Bard. Algebraic cryptanalysis of the data encryption standard. In Steven D. Galbraith, editor, *Cryptography and Coding*, volume 4887 of *Lecture Notes in Computer Science*, pages 152–169. Springer Berlin Heidelberg, 2007.
8. J. Daemen and V. Rijmen. AES proposal: Rijndael, 1998.
9. S. Dawson. Code hopping decoder using a PIC16C56. Microchip confidential, leaked online in 2002, 1998. `http://read.pudn.com/downloads42/sourcecode/embed/144285/keeloq/MCSLRN/DS652B_C.PDF`.
10. Intel Corporation. Intel Turbo Boost Technology in Intel Core Microarchitecture (Nehalem) Based Processors. Technical report, November 2008. `http://download.intel.com/design/processor/applnots/320354.pdf`.
11. Dejan Jovanović and Predrag Janičić. Logical analysis of hash functions. In Bernhard Gramlich, editor, *Frontiers of Combining Systems*, volume 3717 of *Lecture Notes in Computer Science*, pages 200–215. Springer Berlin Heidelberg, 2005.
12. P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *CRYPTO*, pages 388–397, 1999.
13. S. Mangard. A simple power-analysis (SPA) attack on implementations of the AES key expansion. In P. J. Lee and C. H. Lim, editors, *ICISC 2002*, volume 2587 of *LNCS*, pages 343–358. Springer, 2002.
14. S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards (Advances in Information Security)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.
15. V. Manquinho and O. Roussel. Pseudo-boolean competition 2009. Online, http://www.cril.univ-artois.fr/PB09/, July 2009.
16. F. Massacci and L. Marraro. Logical cryptanalysis as a SAT problem. *J. Autom. Reason.*, 24(1-2):165–203, 2000.
17. Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.
18. Ilya Mironov and Lintao Zhang. Applications of SAT solvers to cryptanalysis of hash functions. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006*, volume 4121 of *Lecture Notes in Computer Science*, pages 102–115. Springer Berlin Heidelberg, 2006.
19. Mohamed Saied Emam Mohamed, Stanislav Bulygin, Michael Zohner, Annelie Heuser, Michael Walter, and Johannes Buchmann. Improved algebraic side-channel attack on aes. *J. Cryptographic Engineering*, 3(3):139–156, 2013.
20. National Institute of Standards and Technology . *FIPS PUB 197: Announcing the Advanced Encryption Standard (AES)*. Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD, 2001.
21. National Institute of Standards and Technology. *FIPS PUB 46-3: Data Encryption Standard (DES)*. National Institute for Standards and Technology, Gaithersburg, MD, USA, October 1999.
22. Yossef Oren, Mario Kirschbaum, Thomas Popp, and Avishai Wool. Algebraic Side-Channel Analysis in the Presence of Errors. In *CHES*, pages 428–442, 2010. `http://iss.oy.ne.ro/TASCA`.
23. Yossef Oren, Mathieu Renauld, François-Xavier Standaert, and Avishai Wool. Algebraic side-channel attacks beyond the hamming weight leakage model. In Patrick Schaumont and Emmanuel Prouff, editors, *Workshop on Cryptographic Hardware and Embedded Systems 2012 (CHES 2012), LNCS 7428*, pages 140–154, Leuven, Belgium, 2012. International Association for Cryptologic Research, Springer. `http://iss.oy.ne.ro/Template-TASCA`.
24. 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*, HASP '13, pages 7:1–7:8, New York, NY, USA, 2013. ACM.

25. Yossef Oren and Avishai Wool. TASCA-on-keeloq pseudo-boolean instances. Online, 2010. http://iss.oy.ne.ro/TASCA/Instances.
26. Yossef Oren and Avishai Wool. Template TASCA pseudo-boolean instances. Online, 2012. http://iss.oy.ne.ro/Template-TASCA/Instances.
27. 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 2009*, volume 5747 of *LNCS*, pages 97–111. Springer, 2009.
28. M. Renauld and F.X. Standaert. Algebraic Side-Channel Attacks. In Dongdai Lin Jiwu Jing Feng Bao, Moti Yung, editor, *Information Security and Cryptology (INSCRYPT) 2009*, volume 6151 of *Lecture Notes in Computer Science*, pages 393–410. Springer, 12 2009.
29. H. Satyanarayana. AES128 package. Online, December 2004. http://opencores.net/project,aes_crypto_core.
30. Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer Berlin Heidelberg, 2009.
31. Xinjie Zhao, Tao Wang, Shize Guo, Fan Zhang, Zhijie Shi, Huiying Liu, and Kehui Wu. SAT based Error Tolerant Algebraic Side-Channel Attacks. 2011 Conference on Cryptographic Algorithms and Cryptographic Chips (CASC2011), July 2011.

Appendix: A sample TASCA Instance

The appendix demonstrates some of the equations used in a TASCA attack on AES with set size $k = 3$, following the notation introduced in Subsection 5.1. We show a sample of the goal function, the plaintext assignment, the round functions and the measurement equations. The equations are given in the OPB format supported by the SCIP solver [4]. Full instances can be downloaded from our website at [26].

```
* Plaintext Assignment: 16 equations of 8 bits each
* This compact form of assignment, which specifies 8 bits at a time,
* results in smaller instance sizes and
* better performance using SCIP, when compared to
* assigning a single bit per clause.
* s_0_0_[0:7] = 0xC5:
+1 s_0_0_0 +2 s_0_0_1 +4 s_0_0_2 +8 s_0_0_3 +16 s_0_0_4 ...
  +32 s_0_0_5 +64 s_0_0_6 +128 s_0_0_7 = 197 ;

[...]

* Round 1 of AES: up to 10 sets of equations (typically 1)
* s_1_[0:15]_[0:7] = AddKey(s_0_[0:15]_[0:7], k_[0:15]_[0:7]):
* s_1_0_0 = XOR(s_0_0_0, k_0_0)
 -1 s_1_0_0 +1 s_0_0_0 +1 k_0_0 -2 s_0_0_0 k_0_0 = 0 ;
* s_1_0_1 = XOR(s_0_0_1, k_0_1)
-1 s_1_0_1 +1 s_0_0_1 +1 k_0_1 -2 s_0_0_1 k_0_1 = 0 ;

[...]
```

```
* s_1_15_7 = XOR(s_0_15_7, k_15_7)
-1 s_1_15_7 +1 s_0_15_7 +1 k_15_7 -2 s_0_15_7 k_15_7 = 0 ;

* s_2_0_[0..7] = SubBytes(s_1_0_[0..7]):
-1 s_2_0_0 +1 ~s_1_0_0 ~s_1_0_1 ~s_1_0_2 ~s_1_0_3 ~s_1_0_4 ...
  ~s_1_0_5 ~s_1_0_6 ~s_1_0_7 +1 ~s_1_0_0 s_1_0_1 ~s_1_0_2 ...
  ~s_1_0_3 ~s_1_0_4 ~s_1_0_5 ~s_1_0_6 ~s_1_0_7 ...
   +1 ~s_1_0_0 s_1_0_1 s_1_0_2 s_1_0_3 s_1_0_4 s_1_0_5 s_1_0_6 s_1_0_7 = 0 ;

[...]

* s_4_[0:15]_[0:7] = ShiftRows+MixColumns(s_2_[0:15]_[0:7]) [8-bit]:

* [s_4_0, s_4_1, s_4_2, s_4_3]_[0:7] = ...
  ColumnXform([s_2_0, s_2_5, s_2_10, s_2_15]_[0:7]) [8-bit]:
* s_4_0_0_1_0 = XOR(s_2_0_0, s_2_5_0, s_2_10_0, s_2_15_0)
-1 s_4_0_0_1_0 -8 s_2_0_0 s_2_5_0 s_2_10_0 s_2_15_0  ...
  +4 s_2_0_0 s_2_5_0 s_2_10_0  +4 s_2_0_0 s_2_5_0 s_2_15_0 ...
  +4 s_2_0_0 s_2_10_0 s_2_15_0  +4 s_2_5_0 s_2_10_0 s_2_15_0 ...
  -2 s_2_0_0 s_2_5_0  -2 s_2_0_0 s_2_10_0  -2 s_2_0_0 s_2_15_0 ...
  -2 s_2_5_0 s_2_10_0  -2 s_2_5_0 s_2_15_0  -2 s_2_10_0 s_2_15_0 ...
   +1 s_2_0_0  +1 s_2_5_0 +1 s_2_10_0  +1 s_2_15_0 = 0 ;

[...]

* Side channel measurements:
* Measured Hamming weight for byte 0 of subround 0 = 4:
+1 s_0_0_0 +1 s_0_0_1 +1 s_0_0_2 +1 s_0_0_3 +1 s_0_0_4 +1 s_0_0_5 +1 s_0_0_6 ...
  +1 s_0_0_7 +1 e_s_0_0_p -1 e_s_0_0_n = 4 ;
* Measured Hamming weight for byte 1 of subround 0 = 5:
+1 s_0_1_0 +1 s_0_1_1 +1 s_0_1_2 +1 s_0_1_3 +1 s_0_1_4 +1 s_0_1_5 +1 s_0_1_6 ...
  +1 s_0_1_7 +1 e_s_0_1_p -1 e_s_0_1_n = 5 ;

[...]

* Measured Hamming weight for byte 15 of subround 3 = 0:
+1 s_3_15_0 +1 s_3_15_1 +1 s_3_15_2 +1 s_3_15_3 +1 s_3_15_4 +1 s_3_15_5 ...
  +1 s_3_15_6 +1 s_3_15_7 +1 e_s_3_15_p -1 e_s_3_15_n = 0 ;

* Goal term:
* each side channel introduces two error variables:
* "_p" for positive, and "_n" for negative error.
* The goal function minimises the sum of all these variables.
min: +1 e_s_0_0_p +1 e_s_0_0_n +1 e_s_0_1_p +1 e_s_0_1_n ...
   +1 e_k_15_p +1 e_k_15_n;
```