Cold Boot Key Recovery by Solving Polynomial Systems with Noise

Martin Albrecht* and Carlos Cid

[1] INRIA, Paris-Rocquencourt Center, SALSA Project
UPMC Univ Paris 06, UMR 7606, LIP6, F-75005, Paris, France
CNRS, UMR 7606, LIP6, F-75005, Paris, France
malb@lip6.fr
[2] Information Security Group,
Royal Holloway, University of London
Egham, Surrey TW20 0EX, United Kingdom
carlos.cid@rhul.ac.uk

Abstract. A method for extracting cryptographic key material from DRAM used in modern computers has been recently proposed in [9]; the technique was called *Cold Boot attacks*. When considering block ciphers, such as the AES and DES, simple algorithms were also proposed in [9] to recover the cryptographic key from the observed set of round subkeys in memory (computed via the cipher's key schedule operation), which were however subject to errors due to memory bits decay. In this work we extend this analysis to consider key recovery for other ciphers used in Full Disk Encryption (FDE) products. Our algorithms are also based on closest code word decoding methods, however apply a novel method for solving a set of non-linear algebraic equations with noise based on Integer Programming. This method should have further applications in cryptology, and is likely to be of independent interest. We demonstrate the viability of the Integer Programming method by applying it against the Serpent block cipher, which has a much more complex key schedule than AES. Furthermore, we also consider the Twofish key schedule, to which we apply a dedicated method of recovery.

1 Introduction

The structure of block cipher key schedules has received much renewed attention, since the recent publication of high-profile attacks against the AES [4] and Kasumi [3] in the related-key model. While the practicality of such attacks is subject of debate, they clearly highlight the relevance of the (often-ignored) key schedule operation from a cryptanalysis perspective. An unrelated technique, called *Cold Boot* attacks, was proposed in [9] and also provided an insight into the strength of a particular key schedule against some forms of practical attacks.

* This author was supported by the Royal Holloway Valerie Myerscough Scholarship.

The method is based on the fact that DRAM may retain large part of its content for several seconds after removing its power, with gradual loss over that period. Furthermore, the time of retention can be potentially increased by reducing the temperature of memory. Thus contrary to common belief, data may persist in memory for several minutes after removal of power, subject to slow decay. As a result, data in DRAM can be used to recover potentially sensitive information, such as cryptographic keys, passwords, etc. A typical application is to defeat the security provided by disk encryption programs, such as Truecrypt [16]. In this case, cryptographic key material is maintained in memory, for transparent encryption and decryption of data. One could apply the method from [9] to obtain the computer's memory image, potentially extract the encryption key and then recover encrypted information.

The *Cold Boot* attack has thus three stages: (a) the attacker physically removes the computer's memory, potentially applying cooling techniques to reduce the memory bits decay, to obtain the memory image; (b) locate the cryptographic key material and other sensitive information in the memory image (likely to be subject to errors due to memory bits decay); and (c) recover the original cryptographic key from this information in memory. While all stages present the attacker with several challenges, from the perspective of the cryptologist the one that poses the most interesting problems is the latter stage; we therefore concentrate in this work on stage (c). We refer the reader to [9, 10] for discussion on stages (a) and (b).

A few algorithms were proposed in [9] to tackle stage (c), which requires one to recover the original key based on the observed key material, probably subject to errors (the extent of which will depend on the properties of the memory, lapsed time from removal of power, and temperature of memory). In the case of block ciphers, the key material extracted from memory is very likely to be a set of round subkeys, which are the result of the cipher's key schedule operation. Thus the key schedule can be seen as an *error-correcting code*, and the problem of recovering the original key can be essentially described as a decoding problem.

The paper [9] contains methods for the AES and DES block ciphers (besides discussion for the RSA cryptosystem, which we do not consider in this work). For DES, recovering the original 56-bit key is equivalent to decoding a repetition code. Textbook methods are used in [9] to recover the encryption key from the *closest code word* (i.e. valid key schedule). The AES key schedule is not as simple as DES, but still contains a large amount of linearity (which has also been exploited in recent related-key attacks, e.g. [4]). Another feature is that the original encryption key is used as the initial whitening subkey, and thus should be present in the key schedule. The authors of [9] model the memory decay as a binary asymmetric channel, and recover an AES key up to error rates of $\delta_0 = 0.30, \delta_1 = 0.001$ (see notation in Section 2 below). The results against the AES – under the same model – were further improved in [17, 11].

Contribution of this paper. We note that other block ciphers were not considered in [9]. For instance, the popular FDE product Truecrypt [16] provides the user with a choice of three block ciphers: SERPENT [2], Twofish [14] (both formerly AES candidates) and AES. The former two ciphers present much more complex key schedule operations than DES and AES. Another feature is that the original encryption key does not *explicitly* appear in the expanded key schedule material (but rather has its bits non-linearly combined to derive the several round subkeys). These two facts led to the belief that these ciphers were not susceptible to the attacks in [9], and could perhaps provide an *inherently* more secure alternative to AES when protecting against *Cold Boot* attacks[1].

In this work, we extend the analysis from [9] and demonstrate that one can also recover the encryption key for the SERPENT and Twofish ciphers up to some reasonable amount of error. We propose generic algorithms which apply a novel method for solving a set of non-linear algebraic equations with noise based on Integer Programming[2]. Our methods also allow us to consider different noise models and are not limited to the binary asymmetric channel setting usually considered in the *Cold Boot* scenario; in particular we improve the results against the AES from [9] and extend the range of scenarios in which the attack can be applied when compared to [9, 17, 11]. Finally, we note that our methods can in principle be applied to any cipher and thus provide a *generic* (but possibly impractical for some ciphers) solution to the *Cold Boot* problem.

2 The *Cold Boot* Problem

Cold Boot attacks were proposed and discussed in detail in the seminal work [9]. The authors of [9] noticed that bit decay in DRAM is usually asymmetric: bit flips $0 \to 1$ and $1 \to 0$ occur with different probabilities, depending on the "ground state". To motivate our work, we model more formally the cold boot problem for block ciphers below.

We define the *Cold Boot* problem (for block cipher) as follows. Consider an efficiently computable vectorial Boolean function $\mathcal{KS} : \mathbb{F}_2^n \to \mathbb{F}_2^N$ where $N > n$, and two real numbers $0 \leq \delta_0, \delta_1 \leq 1$. Let $K = \mathcal{KS}(k)$ be the image for some $k \in \mathbb{F}_2^n$, and K_i be the i-th bit of K. Now given K, compute $K' = (K'_0, K'_1, \ldots, K'_{N-1}) \in$

[1] In fact, a message in one of the most popular mailing lists discussing cryptography, commenting at the time about the contributions of [9]: "While they did have some success with recovering an entire AES key schedule uncorrupted, it seems important to note that the simplistic nature of the AES and DES key schedules allowed them to recover the entire original key even after the state had been somewhat degraded with only moderate amounts of work. A cipher with a better key schedule (Blowfish or Serpent, for instance) would seem to offer some defense here"[12], was one of the initial motivations for our work in this problem.

[2] We note that a similar method for key recovery on a different model of leakage, namely side-channel analysis, was independently proposed in [13].

$\mathbb{F}_2^N$ according to the following probability distribution:

$$\begin{array}{ll} Pr[K'_i = 0 \mid K_i = 0] = 1 - \delta_1 \ , & Pr[K'_i = 1 \mid K_i = 0] = \delta_1, \\ Pr[K'_i = 1 \mid K_i = 1] = 1 - \delta_0 \ , & Pr[K'_i = 0 \mid K_i = 1] = \delta_0. \end{array}$$

Thus we can consider such a K' as the output of $\mathcal{KS}$ for some $k \in \mathbb{F}_2^n$ except that K' is noisy, with the probability of a bit 1 in K flipping to 0 is δ_0 and the probability of a bit 0 in K flipping to 1 is δ_1. It follows that a bit $K'_i = 0$ of K' is correct with probability

$$Pr[K_i = 0 \mid K'_i = 0] = \frac{Pr[K'_i = 0 | K_i = 0] Pr[K_i = 0]}{Pr[K'_i = 0]} = \frac{(1 - \delta_1)}{(1 - \delta_1 + \delta_0)}.$$

Likewise, a bit $K'_i = 1$ of K' is correct with probability $\frac{(1-\delta_0)}{(1-\delta_0+\delta_1)}$. We denote these values by Δ_0 and Δ_1 respectively.

Now assume we are given a description of the function $\mathcal{KS}$ and a vector $K' \in \mathbb{F}_2^N$ obtained by the process described above. Furthermore, we are also given a control function $\mathcal{E} : \mathbb{F}_2^n \rightarrow \{True, False\}$ which returns *True* or *False* for a candidate k. The task is to recover k such that $\mathcal{E}(k)$ returns $True$. For example, $\mathcal{E}$ could use the encryption of some known data to check whether k is the original key.

In the context of this work, we can consider the function $\mathcal{KS}$ as the key schedule operation of a block cipher with n-bit keys. The vector K is the result of the key schedule expansion for a key k, and the noisy vector K' is obtained from K due to the process of memory bit decay. We note that in this case, another goal of the adversary could be recovering K rather than k (that is, the expanded key rather than the original encryption key), since with the round subkeys one could implement the encryption/decryption algorithm. In most cases, one should be able to efficiently recover the encryption key k from the expanded key K. However it could be conceivable that for a particular cipher with a highly non-linear key schedule, the problems are not equivalent.

Finally, we note that the *Cold Boot* problem is equivalent to decoding (potentially non-linear) binary codes with biased noise.

3 Block Cipher Key Expansion

In this section we briefly describe some of the relevant features of the key schedule operation of the target ciphers.

3.1 AES

For details of the key schedule of the AES block cipher we refer the reader to [7]. In this work, we are interested in its description as a system of polynomial equations over $\mathbb{F}_2$, see [6]. We note that the non-linearity of the key schedule is provided by four S-box operations in the computation of each round subkey. The explicit degree of the S-box Boolean functions is 7, while it is well known that the key schedule can be described as a system of quadratic equations.

3.2 Serpent

Serpent, designed by Anderson et al. [2], was one of the five AES finalists. The cipher key schedule operation produces 132 32-bit words of key material as follows. First, the user-supplied key k is padded to 256 bits using known constants, and written as eight 32-bit words $w_{-8}, \ldots, w_{-1}$. This new string is then expanded into the prekey words $w_0, \ldots, w_{131}$ by the following affine recurrence:

$$w_i = (w_{i-8} \oplus w_{i-5} \oplus w_{i-3} \oplus w_{i-1} \oplus \psi \oplus i) \lll 11,$$

where ψ is some known constant. Finally the round keys are calculated from the prekeys w_i using the S-boxes S_i in bitslice mode in the following way:

$$\begin{aligned}
\{k_0, k_1, k_2, k_3\} &= S_3(w_0, w_1, w_2, w_3)\\
\{k_4, k_5, k_6, k_7\} &= S_2(w_4, w_5, w_6, w_7)\\
\{k_8, k_9, k_{10}, k_{11}\} &= S_1(w_8, w_9, w_{10}, w_{11})\\
\{k_{12}, k_{13}, k_{14}, k_{15}\} &= S_0(w_{12}, w_{13}, w_{14}, w_{15})\\
\{k_{16}, k_{17}, k_{18}, k_{19}\} &= S_7(w_{16}, w_{17}, w_{18}, w_{19})\\
&\cdots\\
\{k_{124}, k_{125}, k_{126}, k_{127}\} &= S_4(w_{124}, w_{125}, w_{126}, w_{127})\\
\{k_{128}, k_{129}, k_{130}, k_{131}\} &= S_3(w_{128}, w_{129}, w_{130}, w_{131}).
\end{aligned}$$

The rounds subkeys are then $K_i = \{k_{4i}, k_{4i+i}, k_{4i+2}, k_{4i+3}\}$.

We note the following features of the cipher key schedule which are of relevance to *Cold Boot* key recovery: the user-supplied key does not appear in the output of the Serpent key schedule operation, the explicit degree of the S-box Boolean functions is three, and every output bit of the key schedule depends non-linearly on the user-supplied key.

3.3 Twofish

Twofish, designed by Schneier et al. [14], was also one of the five AES finalists. The cipher is widely deployed, e.g. it is part of the cryptographic framework in the Linux kernel and is also available in Full Disk Encryption products. Twofish has a rather complicated key schedule, which makes it a challenging target for *Cold Boot* key recovery attacks. We note that while Twofish is defined for all key sizes up to 256 bits, we will focus here on the 128-bit version. We also follow the notation from [14].

The Twofish key schedule operation generates 40 32-bit words of expanded key $K_0, \ldots, K_{39}$, as well as four key-dependent S-boxes from the user-provided key M. Let $k = 128/64 = 2$, then the key M consists of $8k = 16$ bytes $m_0, \ldots, m_{8k-1}$. The cipher key schedule operates as follows. Each four consecutive bytes are converted into 32-bit words in little endian byte ordering. That is, the leftmost byte is considered as the least significant byte of the 32-bit word. This gives rise to four words M_i. Two key vectors M_e and M_o are defined as $M_e = (M_0, M_2)$ and $M_o = (M_1, M_3)$. The subkey words K_{2i} and K_{2i+1} for $0 \leq i < 20$ are then computed from M_e and M_o by the routine `gen_subkeys` given in Algorithm 2 in the Appendix.

Algorithm 1 (also in the Appendix) defines the function h used in the key schedule. There, we have that q_0 and q_1 are applications of two 8-bit S-boxes defined in [14] and $MDS(Z)$ is a multiplication of Z interpreted as a 4 element vector over the field $\mathbb{F}_{2^8} \cong \mathbb{F}_2[x]/\langle x^8 + x^6 + x^5 + x^3 + 1\rangle$ by a 4×4 MDS matrix. The explicit degree of the S-boxes' Boolean functions is also seven.

Finally, a third vector S is also derived from the key. This is done by combining the key bytes into groups of eight (e.g. $m_0, \ldots, m_7$), interpreting them as a vector over the field $\mathbb{F}_{2^8} \cong \mathbb{F}_2[x]/\langle x^8 + x^6 + x^3 + x^2 + 1\rangle$, which is multiplied by a 4×8 matrix RS. Each resulting four bytes are then interpreted as a 32-bit word S_i. These words make up the third vector $S = (S_1, S_0)$. The key-dependent S-Box g maps 32 bits to 32 bits and is defined as $g(X) = h(X, S)$.

Full disk encryption products use infrequent re-keying, and to provide efficient and transparent access to encrypted data, applications will in practice precompute the key schedule and store the expanded key in memory. For the Twofish block cipher, this means that the subkey words $K_0, \ldots, K_{39}$ as well as the key dependent S-boxes are typically precomputed.

Storing 40 words $K_0, \ldots, K_{39}$ in memory is obviously straightforward (we note however that this set of words does not contain a copy of the user-supplied key). To store the key dependent S-box, the authors of [14] state: "Using 4 Kb of table space, each S-box is expanded to a 8-by-32-bit table that combines both the S-box lookup and the multiply by the column of the MDS matrix. Using this option, a computation of g consists of four table lookups, and three XORS. Encryption and decryption speeds are constant regardless of key size." We understand that most software implementations choose this strategy to represent the S-box (for instance, the Linux kernel chooses this approach, and by default Truecrypt also implements this technique, which can however be disabled with the C macro `TC_MINIMIZE_CODE_SIZE`); we assume this is the case in our analysis.

4 Solving Systems of Algebraic Equations with Noise

In this section we model a new family of problems – solving systems of multivariate algebraic equations with noise – and propose a first method for solving problems from this family. We use the method to implement a *Cold Boot* attack against ciphers with key schedule with a higher degree of non-linearity, such as Serpent.

Polynomial system solving (*PoSSo*) is the problem of finding a solution to a system of polynomial equations over some field $\mathbb{F}$. We consider the set $F = \{f_0, \ldots, f_{m-1}\} \subset \mathbb{F}[x_0, \ldots, x_{n-1}]$. A solution to F is any point $x \in \mathbb{F}^n$ such that $\forall f \in F$, we have $f(x) = 0$. Note that we restrict ourselves to solutions in the base field in the context of this work.

Moreover, denote by *Max-PoSSo* the problem of finding any $x \in \mathbb{F}^n$ that satisfies the maximum number of polynomials in F. Likewise, by *Partial Max-PoSSo* we denote the problem of finding a point $x \in \mathbb{F}^n$ such that for two sets of polynomials $\mathcal{H}, \mathcal{S} \subset \mathbb{F}[x_0, \ldots, x_{n-1}]$, we have $f(x) = 0$ for all $f \in \mathcal{H}$, and the number of polynomials $f \in \mathcal{S}$ with $f(x) = 0$ is maximised. Max-PoSSo is Partial Max-PoSSo with $\mathcal{H} = \varnothing$.

Finally, by *Partial Weighted Max-PoSSo* we denote the problem of finding a point $x \in \mathbb{F}^n$ such that $\forall f \in \mathcal{H} : f(x) = 0$ and $\sum_{f \in \mathcal{S}} \mathcal{C}(f, x)$ is minimised where $\mathcal{C} : f \in \mathcal{S}, x \in \mathbb{F}^n \rightarrow \mathbb{R}_{\geq 0}$ is a cost function that returns 0 if $f(x) = 0$ and some value $v > 0$ if $f(x) \neq 0$. Partial Max-PoSSo is Partial Weighted Max-PoSSo where $\mathcal{C}(f, x)$ returns 1 if $f(x) \neq 0$ for all f.

The family of "Max-PoSSo" problems defined above is analogous to the well-known Max-SAT family of problems. In fact, these problems could well be reduced to their SAT equivalents. However, the modelling as polynomial systems seems more natural in this context since more algebraic structure can be preserved.

4.1 Cold Boot as Partial Weighted Max-PoSSo

We can consider the *Cold Boot* Problem as a Partial Weighted Max-PoSSo problem over $\mathbb{F}_2$. Let $F_\mathcal{K}$ be an equation system corresponding to $\mathcal{KS}$ such that the only pairs (k, K) that satisfy $F_\mathcal{K}$ are any $k \in \mathbb{F}_2^n$ and $K = \mathcal{K}(k)$. In our task however, we need to consider $F_\mathcal{K}$ with k and K'. Assume that for each noisy output bit K'_i there is some $f_i \in F_\mathcal{K}$ of the form $g_i + K'_i$ where g_i is some polynomial. Furthermore assume that these are the only polynomials involving the output bits ($F_\mathcal{K}$ can always be brought into this form) and denote the set of these polynomials by $\mathcal{S}$. Denote the set of all remaining polynomials in $F_\mathcal{K}$ as $\mathcal{H}$, and define the cost function $\mathcal{C}$ as a function which returns

$$\begin{array}{cr} \frac{1}{1-\Delta_0} & \text{for } K'_i = 0, f(x) \neq 0, \\ \frac{1}{1-\Delta_1} & \text{for } K'_i = 1, f(x) \neq 0, \\ 0 & \text{otherwise.} \end{array}$$

Finally, let $F_\mathcal{E}$ be an equation system that is only satisfiable for $k \in \mathbb{F}_2^n$ for which $\mathcal{E}$ returns *True*. This will usually be an equation system for one or more encryptions. Add the polynomials in $F_\mathcal{E}$ to $\mathcal{H}$. Then $\mathcal{H}, \mathcal{S}, \mathcal{C}$ define a Partial

Weighted Max-PoSSo problem. Any optimal solution x to this problem is a candidate solution for the *Cold Boot* problem.

In order to solve Max-PoSSo problems, we propose below an approach which appears to better capture the algebraic structure of the underlying problems (compared to SAT-solvers), and should thus have further applications.

4.2 Mixed Integer Programming

Integer optimisation deals with the problem of minimising (or maximising) a function in several variables subject to linear equality and inequality constraints, and integrality restrictions on some or all the variables. A linear mixed integer programming problem (MIP) is defined as a problem of the form

$$\min_x \{c^T x | Ax \leq b, x \in \mathbb{Z}^k \times \mathbb{R}^l\},$$

where c is an n-vector ($n = k + l$), b is an m-vector and A is an $m \times n$-matrix. This means that we minimise the linear function $c^T x$ (the inner product of c and x) subject to linear equality and inequality constraints given by A and b. Additionally $k \geq 0$ variables are restricted to integer values while $l \geq 0$ variables are real-valued. The set S of all $x \in \mathbb{Z}^k \times \mathbb{R}^l$ that satisfy the linear constraints $Ax \leq b$, that is

$$S = \{x \in \mathbb{Z}^k \times \mathbb{R}^l \mid Ax \leq b\},$$

is called the feasible set. If $S = \varnothing$ the problem is infeasible. Any $x \in S$ that minimises $c^T x$ is an optimal solution.

Efficient MIP solvers use a branch-and-cut algorithm as one of their core components. The main advantage of MIP solvers compared to other branch-and-cut solvers (e.g. SAT-solvers) is that they can relax the problem to a floating point linear programming problem in order to obtain lower and upper bounds. These bounds can then be used to cut search branches. This relaxation also allows one to prove optimality of a solution without exhaustively searching for all possible solutions.

Moreover we can convert the PoSSo problem over $\mathbb{F}_2$ to a mixed integer programming problem using the Integer Adapted Standard Conversion [5] as follows. Consider the square-free polynomial $f \in \mathbb{F}_2[x_0, \ldots, x_{n-1}]$. We interpret the Boolean equation $f = 0$ as an equation over the integers by replacing XOR by addition and AND by multiplication. All solutions of f over $\mathbb{F}_2$ will correspond to multiples of 2 when considered over the integers. Let ℓ be the mininum and u the maximum value of these multiples of two. We introduce an integer variable m and restrict it between $\frac{\ell}{2}$ and $\frac{u}{2}$ (inclusive). Finally we linearise $f - 2m$ and add equations relating the new linear variables to the original monomials. More details of this modelling[3] can be found in [5]. It then follows that solving the

[3] We note that the MIP solver SCIP [1] used in this work generates linear constraints for (among others) AND clauses automatically and thus we do not need to model these explicitly in our experiments.

resulting MIP problem for any objective function will recover a value x that also solves the PoSSo problem.

Moreover we can convert a Partial Weighted Max-PoSSo problem into a Mixed Integer Programming problem as follows. Convert each $f \in \mathcal{H}$ to linear constraints as before. For each $f_i \in \mathcal{S}$ add some new binary slack variable e_i to f_i and convert the polynomial $f_i + e_i$ as before. The objective function we minimise is $\sum c_i e_i$, where c_i is the value of $\mathcal{C}(f, x)$ for some x such that $f(x) \neq 0$. Any optimal solution $x \in S$ will be an optimal solution to the Partial Weighted Max-PoSSo problem.

We note that in our modelling of Cold Boot key recovering as a Mixed Integer Programming problem, we are using a linear objective function, which we expect to be a first order approximation of the true noise model. Our results will however demonstrate that this approximation is sufficient. Finally, we note that the approach discussed above is essentially the non-linear generalisation of decoding random linear codes with linear programming [8].

5 Cold Boot Key Recovery against Block Ciphers

The original approach proposed in [9] is to model the memory decay as a binary asymmetric channel (with error probabilities δ_0, δ_1), and recover the encryption key from the *closest code word* (i.e. valid key schedule) based on commonly used decoding techniques. The model of attack used in [9] often assumes $\delta_1 \approx 0$ (that is, the probability of a 0 bit flipping to 1 is negligible), which appears to be a reasonable assumption to model memory decay in practice. We will sometimes do the same in our discussions below. However, all experimental data in this work was generated with $\delta_1 > 0$, even where our algorithms assume $\delta_1 = 0$, in order to estimate the success rate in practice more precisely. Thus, contrary to prior work, when we construct experimental data we do not consider the asymmetry to be perfect.

Under the adopted model, recovering the original 56-bit key for DES is equivalent to decoding a repetition code, as discussed in [9]. In this section we will discuss potential methods for recovering the user-supplied key for the key schedules of Twofish, Serpent and the AES, under the *Cold Boot* attack scenario. We note that the attack's main parameters (the error probabilities δ_0, δ_1) obviously affect the effectiveness (and the viability) of the methods discussed below; in particular, while some methods may have a superior performance for a certain range of δ_0, δ_1, they may however not be viable outside this particular range. For instance, the technique presented in [11] relies on $\delta_1 = 0$.

5.1 Prior Work on AES

The AES key schedule is not as simple as the one from DES, but still contains a large amount of linearity. Furthermore, the original encryption key is used as the initial whitening subkey, and thus should be present in the key schedule output. The method proposed in [9] for recovering the key for the AES-128 divides this initial subkey into four subsets of 32 bits, and uses 24 bits of the second subkey as redundancy. These small sets are then decoded in order of likelihood, combined and the resulting candidate keys are checked against the full schedule. The idea can be easily extended to the AES with 192- and 256-bit keys. The authors of [9] recover up to 50% of keys for error rates of $\delta_0 = 0.30, \delta_1 = 0$ within 20 minutes. In [17] an improved algorithm making better use of the AES key schedule structure was proposed which allows one to recover the vast majority of keys within 20 minutes for $\delta_0 = 0.70, \delta_1 = 0$. It is noted in [17] that the algorithm can be adapted for the case $\delta_1 > 0$.

In [11] an alternative algorithm is proposed which models the *Cold Boot* problem as a SAT problem by ignoring all output bits equal to zero. In the considered model this implies that the remaining output bits are correct (since $\delta_1 = 0$ implies $\Delta_1 = 1$). Thus a set of correct SAT clauses can be constructed, which – due to the amount of redundant information available in the AES key schedule – still allows one to recover the encryption key. The authors of [11] report recovery of encryption keys for $\delta_0 = 0.80, \delta_1 = 0$ with an average running time of about 30 minutes.

Below we discuss the different methods we have considered for cold boot key recovery, and our results when applied to the AES, Serpent and Twofish. We first discuss a naïve decoding technique in order to have a base line to compare our algebraic technique to. Our algebraic technique is then discussed in Section 5.4.

5.2 Generic Combinatorial Approach

Assuming that the cipher key schedule operation is invertible, we can still consider a somewhat naïve combinatorial approach, even when the user-supplied key does not explicitly appear in the expanded key schedule material. In order to recover the full n-bit key, we will consider at least n bits of key schedule output. We assume that bit-flips are biased towards zero with overwhelming probability (i.e., we assume the asymmetry is perfect) and assume the distribution of bits arising in the original key schedule material is uniform. Then for an appropriate n-bit segment K in the noisy key schedule, we can expect approximately $\frac{n}{2} + r$ zeros, where $r = \lceil \frac{n}{2}\delta_0 \rceil$. We have thus to check

$$\sum_{i=0}^{r} \binom{n/2 + r}{i}$$

candidates for the segment K. Each check entails to correct the selected bits, invert the key schedule and verify the candidate for k using for example $\mathcal{E}$. For $n = 128$ and $\delta_0 = 0.15$ we would need to check approximately 2^{40} candidates; for $\delta_0 = 0.30$ we would have to consider approximately 2^{64} candidates; for $\delta_0 = 0.50$

we would have to consider approximately 2^{85} candidates. By focusing the search around the expected error rate, we may be able to improve these times. This approach is applicable to both SERPENT and the AES. However we need to adapt it slightly for Twofish.

5.3 Adapted Combinatorial Approach for Twofish

We recall that for the Twofish key schedule, we assume that the key dependent S-boxes are stored as a lookup table in memory. In fact, each S-box is expanded to a 8-by-32-bit table holding 32-bit values combining both the S-Box lookup and the multiplication by the column of the MDS matrix (see Section 3.3) Thus, we will have in memory a 2-dimensional 32-bit word array $s[4][256]$, where $s[i][j]$ holds the result of the substitution for the input value j for byte position i. The output of the complete S-Box for the word $X = (X_0, X_1, X_2, X_3)$ is $s[0][X_0] \oplus s[1][X_1] \oplus s[2][X_2] \oplus s[3][X_3]$.

Each array $s[i]$ holds the output of an MDS matrix multiplication by the vector X, with three zero entries and all possible values $0 \leq X_i < 256$, with each value occurring only once. Thus, we have exactly 256 possible values for the 32-bit words in $s[i]$ and we can simply adjust each disturbed word to its closest possible word. Furthermore, we do not need to consider all values, we can simply use those values with low Hamming distance to their nearest candidate word but a large Hamming distance to their second best candidate. We can thus implement a simple decoding algorithm to eventually recover an explicit expression for each of the four key-dependent S-boxes.

Using this method, we can recover all bytes of S_0 and S_1. More specifically, if we assume that $\delta_0 = \delta_1$, we can recover the correct S_0, S_1 with overwhelming probability if 30% of the bits have been flipped. If we assume an asymmetric channel ($\delta_1 \approx 0$) then we can recover the correct values for S_0, S_1 with overwhelming probability if 60% of the bits have been flipped. This gives us 64-bit of *information* about the key.

In order to recover the full 128-bit key, we can adapt the combinatorial approach discussed above. In the noise-free case, we can invert the final modular addition and the MDS matrix multiplication. Since these are the only steps in the key schedule where diffusion between S-box rows is performed, we should get eight 8-bit equation systems of the form $C_1 = Q_0(C_0 \oplus M_0) \oplus M_1$, where Q_0 is some S-box application and C_0 and C_1 are known constants. Each such equation restricts the number of possible candidates for M_0, M_1 from 2^{16} to 2^8. Using more than one pair C_0, C_1 for each user-supplied key byte pair M_0, M_1 allows us to recover the unique key. Thus, although the Twofish key schedule is not as easily reversed as the SERPENT or AES key schedule, the final solving step is still very simple. Thus, the estimates given for the combinatorial approach ($\delta_0 = 0.15 \rightarrow 2^{36}$ candidates and $\delta_0 = 0.30 \rightarrow 2^{62}$ candidates) also apply to Twofish.

Alternatively, we may consider one tuple of C_0, C_1 only and add the linear equations for S. This would provide enough information to recover a unique solution; however S does mix bytes from M_0 across S-box rows, which makes the solving step more difficult.

5.4 Algebraic Approach using Max-PoSSo

If the algebraic structure of the key schedule permits, we can model the *Cold Boot* key recovery problem as a Partial (Weighted) Max-PoSSo problem, and use the methods discussed earlier to attempt to recover the user-supplied key or a noise-free version of the key schedule. We applied those methods to implement a *Cold Boot* attack against the AES and SERPENT. We focused on the 128-bit versions of the two ciphers.

For each instance of the problem we performed 100 experiments with randomly generated keys. In the experiments we usually did not consider the full key schedule but rather a reduced number of rounds of the *key schedule* in order to improve the running time of our algorithms. We note however that this does not mean we are attacking a reduced-round version of the algorithm: considering a reduced amount of data (less redundancy) in the key schedule still allows us to recover the *entire* encryption key of the full-round version of the block cipher. The running time is increased when more data is considered due to the increase of terms in the objective function. Furthermore, we did not include equations for $\mathcal{E}$ explicitly. This is again to reduce the amount of data the solver has to consider. Finally, we also considered at times an "aggressive" modelling, where our algorithm assumes $\delta_1 = 0$ instead of $\delta_1 = 0.001$. In this case all values $K'_i = 1$ are considered correct by the algorithm (since $\Delta_1 = 1$), and as a result all corresponding equations are promoted to the set $\mathcal{H}$. We stress, however, that the input data in our experiments was always generated with $\delta_1 > 0$. Thus, increasing the amount of data considered also increases the chances of including an equation for $K'_i = 1$ which is not correct. We note that in the "aggressive" modelling our problem reduces to Partial Max-PoSSo and that the specific weights assigned in the cost function are irrelevant, since all weights are identical.

Running times for the AES and SERPENT using the MIP solver SCIP [1] are given in Tables 1 and 2 respectively. For each cipher dedicated tuning parameters were used and we also made use of advanced features in SCIP such as the support for AND constraints which are not available in other MIP solvers. The column "a" denotes whether we chose the aggressive ("+") or normal ("–") modelling. The column "cutoff t" denotes the time we maximally allowed the solver to run until we interrupted it. The column r gives the success rate, i.e. the percentage of instances we recovered the correct key for.

For the SERPENT key schedule we consider decays up to $\delta_0 = 0.50, \delta_1 = 0.001$. We also give running times and success rates for the AES up to $\delta_0 = 0.50, \delta_1 = 0.001$ in order to compare our approach with previous work. We note that a success rate

N	δ_0	aggr	limit t	r	min t	avg. t	max t
2	0.05	–	3600.00	59%	50.80 s	2124.90 s	3600.00 s
3	0.15	+	60.0s	63%	1.38 s	8.84 s	41.66 s
4	0.15	+	60.0s	70%	1.78 s	11.77 s	59.16 s
4	0.30	+	600.0s	66%	4.81 s	116.07 s	600.00 s
4	0.30	+	3600.0s	69%	4.86 s	117.68 s	719.99 s
4	0.35	+	600.0s	65%	4.66 s	185.14 s	600.00 s
4	0.35	+	3600.0s	68%	4.45 s	207.07 s	1639.55 s
4	0.40	+	600.0s	47%	4.95 s	284.99 s	600.00 s
4	0.40	+	3600.0s	61%	4.97 s	481.99 s	3600.00 s
5	0.40	+	3600.0s	62%	7.72 s	704.33 s	3600.00 s
4	0.50	+	3600.0s	8%	6.57 s	3074.36 s	3600.00 s
4	0.50	+	7200.0s	13%	6.10 s	5882.66 s	7200.00 s

Table 1. AES considering N rounds of key schedule output.

lower than 100% may still allow a successful key recovery since the algorithm can be run using other data from the key schedule if it fails for the first few rounds. Considering later rounds of the key schedule has no performance penalty for the AES, but does decrease the performance for Serpent as indicated in the row $16 \ll 8$ which considers 16 words of key schedule output starting from the 8-th word. Our attacks were implemented using the Sage mathematics software [15].

We note from the results in Table 1 that the Max-PoSSo based method proposed in this work compares favourably to the results in [9]. However, it offers poorer results when compared to the ones in [17, 11]. While there is no prior work to compare Table 2 with, we note that our technique compares favourably to the generic combinatorial approach discussed earlier. Furthermore, our method has some attractive features and flexibility which allow its application to more extended scenarios and in principle to any block cipher[4].

Finally, we note that our technique does not rely on $\delta_1 = 0$ and can thus be applied if perfect asymmetry cannot be assumed. To demonstrate this feature, we give results against Serpent for our technique when considering symmetric noise (i.e., $\delta_0 = \delta_1$) in Table 3. For comparison, for $\delta_0 = \delta_1 = 0.05$ a combinatorial approach similar to Section 5.2 would have to check rougly $\binom{128}{\lceil 0.05 \cdot 128 \rceil} \approx 2^{36.4}$ candidates. In order to make the comparison fair, we aim for a success rate of $\approx 20\%$ and thus only have to consider roughly 1/5 of those candidates. If each of those checks costs at least $15 \cdot 10^{-8}$ seconds – which ammounts to ≈ 390 CPU cycles on a 2.6 Ghz CPU – then the overall running time would be greater than the average running time reported in Table 3.

[4] We note however that this does not imply the technique is *practical* for all block ciphers as demonstrated by the lack of success against Twofish.

N	δ_0	aggr	limit t	r	min t	avg. t	max t
12	0.05	–	600.0s	37%	8.22 s	457.57 s	600.00 s
12	0.15	+	60.0s	84%	0.67 s	11.25 s	60.00 s
16	0.15	+	60.0s	79%	0.88 s	13.49 s	60.00 s
$16 \lll 8$	0.15	+	1800.0s	64%	95.52 s	1089.80 s	1800.00 s
16	0.30	+	600.0s	74%	1.13 s	57.05 s	425.48 s
16	0.50	+	1800.0s	21%	135.41 s	1644.53 s	1800.00 s
16	0.50	+	3600.0s	38%	136.54 s	2763.68 s	3600.00 s

Table 2. Serpent considering $32 \cdot N$ bits of key schedule output

N	$\delta_0 = \delta_1$	limit t	r	min t	avg. t	max t
12	0.01	3600.0	96%	4.60 s	256.46	3600.0 s
12	0.02	3600.0	79%	8.20 s	1139.72	3600.0 s
8	0.03	3600.0	41%	3.81 s	372.85 s	3600.0 s
12	0.03	7200.0	53%	24.57 s	4205.34 s	7200.0 s
12	0.05	3600.0	18%	5.84 s	1921.89 s	3600.0 s

Table 3. Serpent considering $32 \cdot N$ bits of key schedule output (symmetric noise)

6 Conclusion and Discussions

In this paper we followed up from the original work on *Cold Boot* key recovery attacks in [9, 11, 17], and extended the analysis to consider other block ciphers such as Twofish and Serpent. Our algorithms apply a novel method for solving a set of non-linear algebraic equations with noise based on Integer Programming. Besides improving some existing results and extending the range of scenarios in which the attacks can be applied, this paper also brings into attention two topics which in our opinion should be of enough interest for future research in cryptology:

Block Cipher Key Schedule: the structure of the key schedule of block ciphers has recently started attracting much attention from the cryptologic research community. Traditionally, the key schedule operation has perhaps received much less consideration from designers, and other than for efficiency and protection against some known attacks (e.g. slide attacks), the key schedule was often designed in a somewhat ad-hoc way (in contrast to the usually well-justified and motivated cipher round structure). However the recent attacks against the AES and Kasumi have brought this particular operation to the forefront of block cipher cryptanalysis (and as a result, design). While one can argue that some of the models of attack used in the recent related-key attacks may be far too generous to be of practical relevance, it is clear that resistance of ciphers against these attacks will from now on be used as another form of measure of security of block ciphers.

In this spirit, we propose in this paper a further measure of security for key schedule operations, based on the *Cold Boot* attack scenario. These attacks are arguably more practical than some of the other attacks targeting the key schedule operation. More importantly, we believe the model can be used to further evaluate the strength of the key schedule operation of block ciphers. Our results show however that it is not trivial to provide high security against *Cold Boot* attacks. In fact, by proposing generic algorithms for solving the *Cold Boot* problem, we showed that, contrary to general belief, several popular block ciphers are also susceptible to attack under this model. How to come up with design criteria for a secure key schedule under this model (while preserving other attractive features such as efficiency) remains a topic for further research.

Polynomial System Solving with Noise: another contribution of this paper, which is very likely to be of independent interest, is the treatment of the problem of solving non-linear multivariate equations with noise. In fact, several interesting problems in cryptography such as algebraic attacks, side-channel attacks and the cryptanalysis of LPN/LWE-based schemes can be naturally modeled as Max-PoSSo problems. However, so far this problem was not considered in its general form. This paper presents a formalisation of this problem and a novel method, based on Integer Programming, which proved to be a powerful technique in some situations. We expect that this will bring MIP solvers further to the attention of the cryptography research community and consider studying and improving (MIP-based) Max-PoSSo methods an interesting area for future research.

7 Acknowledgements

We would like to thank Stefan Heinz, Timo Berthold and Ambros M. Gleixner for helpful discussions on mixed integer programming and for providing tuning parameters for SCIP suitable for our problems. We would also like to thank Kenny Paterson for helpful comments on this work.

References

1. Tobias Achterberg. Constraint Integer Programming. PhD thesis, TU Berlin 2007. `http://scip.zib.de`
2. Eli Biham, R.J. Anderson, and L.R. Knudsen. SERPENT: A New Block Cipher Proposal. In S. Vaudenay, editor, *Fast Software Encryption 1998*, volume 1372 of *LNCS*, pages 222–238. Springer–Verlag, 1998.
3. Eli Biham, Orr Dunkelman, and Nathan Keller. A Related-Key Rectangle Attack on the Full KASUMI. In , *Advances in Cryptology - ASIACRYPT 2005*, volume 3788 of *LNCS*, pages 443–561. Springer-Verlag, 2009.
4. Alex Biryukov, Dmitry Khovratovich and Ivica Nikolić. Distinguisher and Related-Key Attack on the Full AES-256. *Advances in Cryptology - CRYPTO 2009*, volume 5677 of *LNCS* , pages 231–249, Springer-Verlag 2009.

5. Julia Borghoff, Lars R. Knudsen and Mathias Stolpe. Bivium as a Mixed-Integer Linear Programming Problem. *Cryptography and Coding – 12th IMA International Conference*, volume 5921 of *LNCS*, 133–152, Springer-Verlag 2009.
6. Carlos Cid, Sean Murphy, and Matthew J.B. Robshaw. *Algebraic Aspects of the Advanced Encryption Standard*. Springer–Verlag, 2007.
7. J. Daemen and V. Rijmen. *The Design of Rijndael*. Springer–Verlag, 2002.
8. Jon Feldman. Decoding Error-Correcting Codes via Linear Programming. PhD thesis, Massachusetts Institute of Technology 2003.
9. J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino and Ariel J. Feldman, Jacob Appelbaum and Edward W. Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. *USENIX Security Symposium*, 45–60, USENIX Association 2009.
10. Nadia Heninger and Hovav Shacham. Reconstructing RSA Private Keys from Random Key Bits. *Cryptology ePrint Archive, Report 2008/510*, 2008.
11. Abdel A. Kamal and Amr M. Youssef. Applications of SAT Solvers to AES key Recovery from Decayed Key Schedule Images. *Proceedings of The Fourth International Conference on Emerging Security Information, Systems and Technologies – SECURWARE 2010*, July 18 – 25, 2010 - Venice/Mestre, Italy.
12. J. Lloyd. Re: cold boot attacks on disk encryption. Message posted to *The Cryptography Mailing List* on 21 Feb 2008, archived at `http://www.mail-archive.com/cryptography@metzdowd.com/msg08876.html`
13. Y. Oren, M. Kirschbaum, T. Popp and A. Wool. Algebraic Side-Channel Analysis in the Presence of Errors. *Proceedings of CHES 2010*, LNCS 6225, pp. 428–442, Springer-Verlag 2010.
14. Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall and Niels Ferguson. Twofish: A 128-Bit Block Cipher. `http://www.schneier.com/paper-twofish-paper.pdf`, 1998.
15. William Stein et al. Sage Mathematics Software (Version 4.4.1). The Sage Development Team, 2010, `http://www.sagemath.org`.
16. TrueCrypt Project, `http://www.truecrypt.org/`.
17. Alex Tsow. An Improved Recovery Algorithm for Decayed AES Key Schedule Images. *Proceedings of SAC 2009*, volume 5867 of *LNCS*, pages 215–230, Springer-Verlag 2009.

```
**Input**: $Z$ – a 32-bit word
**Input**: $L$ – a list of two 32-bit words
**Result**: a 32-bit word
**begin**
    $L_0, L_1 \longleftarrow L[0], L[1]$;
    $z_0, z_1, z_2, z_3 \longleftarrow$ split $Z$ into four bytes;
    $z_0, z_1, z_2, z_3 \longleftarrow q_0[z_0], q_1[z_1], q_0[z_2], q1[z_3]$;
    $z_0, z_1, z_2, z_3 \longleftarrow z_0 \oplus L_1[0], z_1 \oplus L_1[1], z_2 \oplus L_1[2], z_3 \oplus L_1[3]$;
    $z_0, z_1, z_2, z_3 \longleftarrow q_0[z_0], q_0[z_1], q_1[z_2], q_1[z_3]$;
    $z_0, z_1, z_2, z_3 \longleftarrow z_0 \oplus L_0[0], z_1 \oplus L_0[1], z_2 \oplus L_0[2], z_3 \oplus L_0[3]$;
    $z_0, z_1, z_2, z_3 \longleftarrow q_1[z_0], q_0[z_1], q_1[z_2], q_0[z_3]$;
    $z_0, z_1, z_2, z_3 \longleftarrow MDS(z_0, z_1, z_2, z_3)$;
    **return** the 32-bit word consisting of the four bytes $z_0, z_1, z_2, z_3$;
**end**
```

Algorithm 1: h

```
**Input**: $i$ – an integer
**Input**: $M_e$ – a list of 32-bit words
**Input**: $M_o$ – a list of 32-bit words
**Result**: two 32-bit words
**begin**
    $\rho \longleftarrow 2^{24} + 2^{16} + 2^8 + 2^0$;
    $A_i \longleftarrow h(2i\rho, M_e)$;
    $B_i \longleftarrow h((2i+1)\rho, M_o) \lll 8$;
    $K_{2i} \longleftarrow A_i + B_i \bmod 2^{32}$;
    $K_{2i+1} \longleftarrow (A_i + 2B_i \bmod 2^{32}) \lll 9$;
    **return** $K_{2i}, K_{2i+1}$;
**end**
```

Algorithm 2: gen_subkeys