A Comparison between Two Off-the-Shelf Algebraic Tools for Extraction of Cryptographic Keys from Corrupted Memory Images

Abdel Alim Kamal[1], Roger Zahno[2], and Amr M. Youssef[2]

[1] Faculty of Computers and Information, Menofia University, Shebin El-kom, Menofia, 32511 Egypt
a_kamala@ci.menofia.edu.eg

[2] Institute for Information Systems Engineering, Concordia University, Montreal, Quebec, H3G 1M8 Canada
{r_zahno,youssef}@ciise.concordia.ca

Abstract. Cold boot attack is a class of side channel attacks which exploits the data remanence property of random access memory (RAM) to retrieve its contents which remain readable shortly after its power has been removed. Specialized algorithms have been previously proposed to recover cryptographic keys of several ciphers from decayed memory images. However, these techniques were cipher-dependent and certainly uneasy to develop and fine tune. On the other hand, for symmetric ciphers, the relations that have to be satisfied between the subround key bits in the key schedule always correspond to a set of nonlinear Boolean equations. In this paper, we investigate the use of an off-the-shelf SAT solver (CryptoMiniSat), and an open source Gröbner basis tool (PolyBoRi) to solve the resulting system of equations. We also provide the pros and cons of both approaches and present some simulation results for the extraction of AES and Serpent keys from decayed memory images using these tools.

1 Introduction

Cryptanalytic attacks can be classified into pure mathematical attacks and side channel attacks. Pure mathematical attacks, are traditional cryptanalytic techniques that rely only on known or chosen input-output pairs of the cryptographic function, and exploit the inner structure of the cipher to reveal secret key information. On the other hand, in side channel attacks, it is assumed that the attacker has some physical access to the cryptographic device through one or more side channel. Well-known side channels, which can leak critical information about the encryption state, include timing information [1] and power consumption [2].

In addition to these commonly exploited side channels, the remanence effect of random access memory (RAM) is a highly critical side channel that has been recently exploited by cold boot attacks [3][4] to retrieve secret keys from RAM. Although dynamic RAMs (DRAMs) become less reliable when its contents are not refreshed, they are not immediately erased. In fact, contrary to popular belief, DRAMs may retain their contents for seconds to minutes after power is lost and even if they are removed from the computer motherboard. A cold boot attack is launched by removing the memory module, after cooling it, from the target system and immediately plugging it in another system under the adversarys control. This system is then booted to access the memory. Another possible approach to execute the attack is to cold boot the target machine by cycling its power off and then on without letting it shut down properly. Upon reboot, a lightweight operating system is instantly booted where the contents of targeted memory are dumped to a file.

Experimental results in [5] show how data are retained for a relatively long time in computer memories after a system power off. However, the first work explicitly exploiting those observations to recover cryptographic keys from the memory was reported by Halderman *et al.* [3] where they presented proof of concept experiments which showed that it is practically feasible to perform cold boot attacks exploiting the remanence effect of RAMs to recover secret keys of DES, AES and RSA. After the publication of Halderman *et al.* [3], several other authors (e.g., [6], [7] [4]) further improved upon this proof of concept and presented algorithms that solved cases with higher decay factors. However, almost all these previously proposed techniques were cipher-dependent and certainly uneasy to develop and fine tune. On the other hand, for symmetric ciphers, the relations that have to be satisfied between the subround key bits in the key schedule always correspond to a set of nonlinear Boolean equations. In this paper, we investigate the use of an off-the-shelf SAT solver (CryptoMiniSat [8]), and an open source Gröbner basis tool (PolyBoRi [9]) to solve the resulting system of equations. We also discuss the pros and cons of both tools and present some experimental results for the extraction of AES and Serpent keys from decayed memory images.

The remainder of the paper is organized as follows. In section 2 we briefly review some related works. SAT solvers and Gröbner basis tools, and their applications in cryptography, are discussed in section 3. Relevant details of the the structure of the AES and Serpent key schedules are discussed in section 4. Our experimental results are presented and discussed in section 5. Finally, our conclusion is given in section 6.

2 Related Work

Cryptographic key recovery from memory or memory dumps, for malicious or forensic purposes, has attracted great attention of security professionals and cryptographic researchers. In [10], Shamir and van Someren considered the problem of locating cryptographic keys hidden in large amount of data, such as the complete file system of a computer system. In addition to efficient algebraic attacks locating secret RSA keys in long bit strings, they also presented more general statistical attacks which can be used to find arbitrary cryptographic keys embedded in large files. This statistical approach relies on the simple fact that good cryptographic keys pose high entropy. Areas with unusually high entropy can be located by searching for unique byte patterns in sliding windows and then selecting those windows with the highest numbers of unique bytes as a potential places for the key. Moe *et al.* [11] developed a proof of concept tool, *Interrogate*, which implements several search methods for a set of key schedules. To verify the effectiveness of the developed tool, they investigated key recovery for systems running in different states (live, screen-saver, dismounted, hibernation, terminated, logged out, reboot, and boot states). Another proof of concept tool, *Disk Decryptor*, which can extract Pretty Good Privacy (PGP) and Whole Disk Encryption (WDE) keys from dumps of volatile memories was presented in [12].

All the above techniques and tools took another dimension after the publication of the cold boot attack by Halderman *et al.* [3]. While the remanence effect of RAM has already been known since decades [5], it attracted greater attention in cryptography only after Halderman *et al.* work in 2008, which explicitly exploited those observations to recover cryptographic keys from the memory. They developed tools which capture everything present in RAM before power was cut off and developed proof of concept tools which can analyze these memory copies to extract secret DES, AES and RSA keys.

In particular, Heninger *et al.* showed that an RSA private key with small public exponent can be efficiently recovered given a 27% fraction of its bits at random. They have also developed a recovery algorithm for the 128-bit version of AES (AES-128) that recovers keys from 30% decayed AES-128 Key Schedule images in less than 20 minutes for half of the simulated cases. Tsow [6] further improved upon the proof of concept in Halderman *et al.* and presented a heuristic algorithm that solved all cases at 50% decay in under half a second. At 60% decay, Tsow recovered the worst case in 35.5 seconds while solving the average case in 0.174 seconds. At the extended decay rate of 70%, recovery time averages grew to over 6 minutes with the median time at about five seconds.

In [7], Albrecht *et al.* proposed methods for key recovery of ciphers (AES, Serpent and Twofish) used in Full Disk Encryption (FDE) products where they applied a method for solving a set of non-linear algebraic equations with noise based on mixed integer programming. To improve the running time of their algorithms, they only considered a reduced number of rounds. Applying their algorithms, they obtained satisfactory success rates for key recovery using the Serpent key schedule up to 30% decay and for the AES up to 40% decay.

3 Modern Algebraic Tools and Their Applications to Cryptography

The use of SAT solvers and Gröbner basis in cryptanalysis has recently attracted the attention of cryptanalysts. Courtois *et al.* [13] demonstrated a weakness in KeeLog by presenting an attack which requires about 2^{32} known plaintexts. For 30% of all keys, the full key can be recovered against a complexity of 2^{28} KeeLoq encryptions. In [14], 6 rounds of DES are attacked with only a single known plaintext/ciphertext pair using a SAT solver. Erickson *et al.* [15] used the

SAT solver and Gröbner basis [16] attacks against SMS4 on equation system over GF(2) and GF(2^8). In [17], a practical Gröbner basis [16] attack using Magma was applied against the ciphers Flurry and Curry, recovering the full cipher key by requiring only a minimal number of plaintext/ciphertext pairs.

SAT solvers and Gröbner basis have also been applied to the cryptanalysis of stream ciphers. Eibach *et al.* [18] presented experimental results on attacking a reduced version of Trivium (Bivium) using exhaustive search, a SAT solver, a binary decision diagram (BDD) based attack, a graph theoretic approach, and Gröbner basis. Their result implies that the usage of the SAT solver is faster than the other attacks. The full key of Hitag2 stream cipher is recovered in a few hours using MiniSat 2.0 [19]. In [20], the full 48-bit key of the MiFareCrypto 1 algorithm was recovered in 200 seconds on a PC, given 1 known initial vector (IV) from one single encryption. In [21], Velichkov *et al.* applied the Gröbner basis on a reduced 16 bit version of the stream cipher Lex.

Mironov and Zhang [22] described some initial results on using SAT solvers to automate certain components in cryptanalysis of hash functions of the MD and SHA families. De *et al.* [23] presented heuristics for solving inversion problems for functions that satisfy certain statistical properties similar to that of random functions. They demonstrate that this technique can be used to solve the hard case of inverting a popular secure hash function and were able to invert MD4 up to 2 rounds and 7 steps in less than 8 hours. In [24], Sugita *et al.* used the Gröbner basis to improve the attack on the 58-round SHA-1 hash function to 2^{31} computations instead of 2^{34} in Wang's method [25].

3.1 Gröbner Basis and PolyBoRi

A Gröbner basis is a set of multivariate polynomials that have desirable algorithmic properties. In what follows, we briefly review some basic definitions and algebraic preliminaries related to Gröbner basis as presented in [26].

Let K be any field (in here we are interested in the case where $K = \mathbb{F}_2$.) We write $K[x_1, ..., x_n]$ for the ring of polynomials in n for the variables x_i having its coefficients in the field K.

Definition 1. *A subset $I \subset K[x_1, ..., x_n]$ is an ideal if it satisfies:*

1. *$0 \in I$.*
2. *if f,$g \in I$, then $f + g \in I$.*
3. *if $f \in I$ and $h \in K[x_1, ..., x_n]$, then $hf \in I$.*

Definition 2. *Let $f_1, ..., f_m$ be polynomials in $K[x_1, ..., x_n]$. Define the ideal $\langle f_1, ..., f_m\rangle = \{\sum_{i=1}^{m} h_i f_i : h_1, ..., h_m \in K[x_1, ..., x_n]\ \}$. If there exists a finite set of polynomials in $K[x_1, ..., x_n]$ that generate the given ideal, we call this set a basis.*

Definition 3. *A monomial ordering on $K[x_1, ..., x_n]$ is any relation $>$ on $\mathbb{Z}^n_{\geq 0}$, or equivalently, any relation on the set of monomials x^α, $\alpha \in \mathbb{Z}^n_{\geq 0}$, satisfying:*

1. *$>$ is a total ordering on $\mathbb{Z}^n_{\geq 0}$.*
2. *if $\alpha > \beta$ and $\alpha, \beta, \gamma \in \mathbb{Z}^n_{\geq 0}$, then $\alpha + \gamma > \beta + \gamma$.*
3. *$>$ is a well ordering on $\mathbb{Z}^n_{\geq 0}$. That is every nonempty subset of $\mathbb{Z}^n_{\geq 0}$ has a smallest element with respect to $>$.*

An example of monomial ordering for our application is lexicographic order which is defined as follows:

Definition 4. *(Lexicographic Order (lex)). Let $\alpha = (\alpha_1, ..., \alpha_n)$, and $\beta = (\beta_1, ..., \beta_n) \in \mathbb{Z}^n_{\geq 0}$. We say $\alpha >_{lex} \beta$ if, in the vector difference $\alpha - \beta \in \mathbb{Z}^n$, the left-most nonzero entry is positive. We will write $x^\alpha >_{lex} x^\beta$ if $\alpha >_{lex} \beta$.*

Definition 5. *Let $f = \Sigma_\alpha a_\alpha x^\alpha$ be a non-zero polynomial in P and let $>$ be a monomial order. The multidegree of f is $multideg(f) = max_>(\alpha \in \mathbb{Z}^n_{\geqslant 0} : a_\alpha \neq 0)$.*

Definition 6. *(leading term of a polynomial). Let $f(x) = \sum_{i=1}^m c_\alpha x^\alpha : c_\alpha \in K$ is non-zero and $>$ is the order relation defined for the monomials of the polynomial $f(x)$. The greatest monomial in $f(x)$, regarding to the order relation $>$, is called the leading monomial for the polynomial $f(x)$ and is represented by $LM(f) = x^{multideg(f)}$. Also the set $M(f)$ consists of all monomials of $f(x)$ and $T(f)$ denote the set of all terms of $f(x)$. The coefficient of the leading monomial is represented by $LC(f) = a_{multideg(f)} \in K$ and called the leading coefficient. The term containing both the leading coefficient and leading monomial is called the leading term, represented by $LT(f) = LC(f) \cdot LM(f)$.*

The idea of Gröbner basis was first proposed by Buchberger [16] to study the membership of a polynomial in the ideal of the polynomial ring.

Definition 7. *(Gröbner basis) Let an ideal I be generated by $G = g_1, ..., g_m$, where g_i, $1 \leq i \leq m$ is a polynomial. G is called the Gröbner basis for the ideal I, if:*

$$\langle LT(I) \rangle = \langle LT(g_1), ..., LT(g_m) \rangle,$$

where $\langle LT(I) \rangle$ denotes the ideal generated by the leading terms of the members in I.

One can view Gröbner basis as a multivariate, non-linear generalization of the Euclidean algorithm for computation of univariate greatest common divisors, Gaussian elimination for linear systems, and integer programming problems. In this work, we use Gröbner basis as an algebraic tool that allows us to solve non-linear Boolean equations by using the PolyBoRi framework.

The following example explains the main involved steps and commands for the PolyBoRi framework in Sage [27] to solve a given system of nonlinear Boolean equations.

Example 8. Consider the following system of non-linear Boolean equations

$$\begin{aligned} x_1x_2 \oplus x_3x_4 &= 1, \\ x_1x_3x_5 \oplus x_4x_5 &= 0, \\ x_1x_2x_5 \oplus x_3x_5 &= 0, \\ x_2x_3 \oplus x_3x_4x_5 &= 1, \end{aligned} \tag{1}$$

```
c    Lines starting with 'c' are comments
c    Step 1 defines the Polynomial Ring; where GF(2) defines the Galois field (GF)
c    of 2 elements as the base ring, 5 is the number of variables and order = 'lex'
c    sets the order to lexical order
c    Step 2 defines the Ideal taking a set of homogeneous equations
c    as calling parameter
c    Step 3 combines the ideal I with the field ideal;
c    limiting the solution range to F2
c    Step 4 executes the Gröbner basis returning the result

sage: PR.<x1,x2,x3,x4,x5> = PolynomialRing(GF(2), 5, order='lex')
sage: I = ideal([x1*x2 + x3*x4 + 1, x1*x3*x5 + x4*x5,
          x1*x2*x5 + x3*x5, x2*x3 + x3*x4*x5 + 1])
sage: J = I + sage.rings.ideal.FieldIdeal(PR)
sage: J.groebner_basis()
```

Fig. 1. Working with PolyBoRi to solve the systems of equations in (1)

Figure 1 shows the steps to be executed in PolyBoRi to solve the Gröbner basis. As shown in the figure, the function *ideal()* in step 2 takes the corresponding homogeneous system of equations as a calling parameter.

The resulting Gröbner basis is given by $[x_1 + x_4 + 1, x_2 + 1, x_3 + 1, x_4^2 + x_4, x_5]$. In this notation, x_i appearing in a separate term by itself implies that the system of equations under consideration can be solved by setting $x_i = 0$. Similarly, $x_i + 1$ implies that $x_i = 1$. Also, the notation $x_i + x_i^2$ implies that x_i can be assigned a 0 or a 1. Thus the above basis corresponds to the following two independent solutions: $\{x_1 = 1, x_2 = 1, x_3 = 1, x_4 = 0, x_5 = 0\}$ and $\{x_1 = 0, x_2 = 1, x_3 = 1, x_4 = 1, x_5 = 0\}$.

3.2 The SAT Problem and CryptoMiniSat

The Boolean satisfiability (SAT) problem [28] is defined as follows: Given a Boolean formula, check whether an assignment of Boolean values to the propositional variables in the formula exists such that the formula evaluates to true. If such an assignment exists, the formula is said to be satisfiable; otherwise, it is unsatisfiable. For a formula with m variables, there are 2^m possible truth assignments. The conjunctive normal form (CNF) is frequently used for representing Boolean formulas. In CNF, the variables of the formula appear in literals (e.g., x) or their negation (e.g., $\overline{x}$). Literals are grouped into clauses, which represent a disjunction (logical OR) of the literals they contain. A single literal can appear in any number of clauses. The conjunction (logical AND) of all clauses represents a formula. For example, the CNF formula $(x_1) \wedge (\overline{x}_2 \vee \overline{x}_3) \wedge (x_1 \vee x_3)$ contains three clauses: x_1, $\overline{x}_2 \vee \overline{x}_3$ and $x_1 \vee x_3$. Two literals in these clauses are positive (x_1, x_3) and two are negative ($\overline{x}_2$, $\overline{x}_3$). For a variable assignment to satisfy a CNF formula, it must satisfy each of its clauses. For example, if x_1 is true and x_2 is false, then all three clauses are satisfied, regardless of the value of x_3.

While the SAT problem has been shown to be NP-complete [28], efficient heuristics exist that can solve many real-life SAT formulations. Furthermore, the wide range of target applications of SAT have motivated advances in SAT solving techniques that have been incorporated into freely-available SAT solvers such as the CryptoMiniSat.

When preparing the input to the SAT solver, the terms of quadratic and higher degree are handled by noting that (for example) the logical expression

$$(x_1 \vee \overline{T})(x_2 \vee \overline{T})(x_3 \vee \overline{T})(x_4 \vee \overline{T})(T \vee \overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3 \vee \overline{x}_4) \tag{2}$$

is tautologically equivalent to $T \Leftrightarrow (x_1 \wedge x_2 \wedge x_3 \wedge x_4)$, or the $GF(2)$ equation $T = x_1x_2x_3x_4$. Similar expressions exist for higher order terms. Thus, the system of equations obtained in this step can be linearized by introducing new variables as illustrated by the following example.

Example 9. Suppose we would like to find the Boolean variable assignment that satisfies the following formula

$$x_0 \oplus x_1x_2 \oplus x_0x_1x_2 = 0.$$

Then, using the approach illustrated in (2), we introduce two linearization variables, $T_0 = x_1x_2$ and $T_1 = x_0x_1x_2$. Thus we have

$$\begin{aligned}
&x_0 \oplus T_0 \oplus T_1 = 0,\\
&(\overline{T}_0 \vee x_1) \wedge (\overline{T}_0 \vee x_2) \wedge (T_0 \vee \overline{x}_1 \vee \overline{x}_2) = 1,\\
&(\overline{T}_1 \vee x_0) \wedge (\overline{T}_1 \vee x_1) \wedge (\overline{T}_1 \vee x_2) \wedge\\
&(T_1 \vee \overline{x}_0 \vee \overline{x}_1 \vee \overline{x}_2) = 1.
\end{aligned} \tag{3}$$

Since the CryptoMiniSAT expects only positive clauses and the CNF form does not have any constants, we need to overcome the problem that the first line in (3) corresponds to a negative, i.e., false, clause. Adding the clause consisting of a dummy variable, d, or equivalently $(d \wedge d \cdots \wedge d)$ would require the variable d to be true in any satisfying solution, since all clauses must be true in any satisfying solution. In other words, the variable d will serve the place of the constant 1.

Therefore, the above formula can be expressed as

$$\begin{aligned}
&d = 1,\\
&x_0 \oplus T_0 \oplus T_1 \oplus d = 1,\\
&(\overline{T}_0 \vee x_1) \wedge (\overline{T}_0 \vee x_2) \wedge (T_0 \vee \overline{x}_1 \vee \overline{x}_2) = 1,\\
&(\overline{T}_1 \vee x_0) \wedge (\overline{T}_1 \vee x_1) \wedge (\overline{T}_1 \vee x_2) \wedge\\
&(T_1 \vee \overline{x}_0 \vee \overline{x}_1 \vee \overline{x}_2) = 1.
\end{aligned}$$

Applying the same logic to the system of equations in (1), we obtain

$$\begin{aligned}
&d = 1,\\
&T_1 \oplus T_2 = 1,\\
&(\overline{T_1} \vee x_1) \wedge (\overline{T_1} \vee x_2) \wedge (T_1 \vee \overline{x_1} \vee \overline{x_2}) = 1,\\
&(\overline{T_2} \vee x_3) \wedge (\overline{T_2} \vee x_4) \wedge (T_2 \vee \overline{x_3} \vee \overline{x_4}) = 1,\\
&T_3 \oplus T_4 \oplus d = 1,\\
&(\overline{T_3} \vee x_1) \wedge (\overline{T_3} \vee x_3) \wedge (\overline{T_3} \vee x_5) \wedge\\
&(T_3 \vee \overline{x_1} \vee \overline{x_3} \vee \overline{x_5}) = 1,\\
&(\overline{T_4} \vee x_4) \wedge (\overline{T_4} \vee x_5) \wedge (T_4 \vee \overline{x_4} \vee \overline{x_5}) = 1,\\
&T_5 \oplus T_6 \oplus d = 1,\\
&(\overline{T_5} \vee x_1) \wedge (\overline{T_5} \vee x_2) \wedge (\overline{T_5} \vee x_5) \wedge\\
&(T_3 \vee \overline{x_1} \vee \overline{x_2} \vee \overline{x_5}) = 1,\\
&(\overline{T_6} \vee x_3) \wedge (\overline{T_6} \vee x_5) \wedge (T_6 \vee \overline{x_3} \vee \overline{x_5}) = 1,\\
&T_7 \oplus T_8 = 1,\\
&(\overline{T_7} \vee x_2) \wedge (\overline{T_7} \vee x_3) \wedge (T_7 \vee \overline{x_2} \vee \overline{x_3}) = 1,\\
&(\overline{T_8} \vee x_3) \wedge (\overline{T_8} \vee x_4) \wedge (\overline{T_8} \vee x_5) \wedge\\
&(T_8 \vee \overline{x_3} \vee \overline{x_4} \vee \overline{x_5}) = 1,
\end{aligned}$$

Figure 2 shows the CryptoMiniSat input file corresponding to the above system of equations. As shown in the figure, a negative number implies that the variables assumes a value = 0 and a positive number implies a value = 1. Lines starting with 'x' denote an XOR equation and each lines is terminated with '0'.

From the above examples, its is clear that, compared to PolyBoRi, preparing the input for the CryptoMiniSat requires relatively longer pre-processing steps. Also, unlike the Gröbner basis approach which returns the general form of the solution, CryptoMiniSat returns one valid solution. To find the other solutions, the already found solutions have to be negated and added to the SAT solver input file. In the example above, the first solution returned by the CryptoMiniSat ({1, −2, 3, 4, 5, −6, −7, 8, −9, −10, −11, −12, 13, −14}) is negated ({−1, 2, −3, −4, −5, 6, 7, −8, 9, 10, 11, 12, −13, 14}) and added to the SAT solver input file as a new entry. When running the SAT solver again, this added entry forces the SAT solver to eliminate this as a possible solution and search for a new one that solves the SAT problem. When doing so, the SAT solver returns the second possible solution ({1, 2, 3, 4, −5, −6, 7, −8, −9, −10, −11, −12, 13, −14}).

4 Structure of the AES-128 and Serpent Key Schedules

In this section, we briefly review the relevant details of the AES-128 and Serpent key schedules.

4.1 Key Schedule of AES-128

In the following we describe the AES-128 key scheduler [29], [30]. AES-128 works with a user key (Master Key) of 128 bits (16 bytes) represented by a 4x4 array

```
c    Lines starting with 'c' are comments
c    The first line in the SAT file is in the form: 'p cnf # variables # clause'
c    Each line ends with '0' and lines starting with 'x' denote XOR equations
c    True variables are denoted by positive numbers and False variables
c    are denoted by negating the number; example: x1 → 2; (x̅2 → -3)
c    d → 1, x1 → 2, x2 → 3, ... , T6 → 12, T7 → 13, T8 → 14

p cnf 14 32
1 0
x 7 8 0
-7 2 0
-7 3 0
7 -2 -3 0
-8 4 0
-8 5 0
8 -4 -5 0
x 9 10 1 0
-9 2 0
-9 4 0
-9 6 0
9 -2 -4 -6 0
-10 5 0
-10 6 0
10 -5 -6 0
x 11 12 1 0
-11 2 0
-11 3 0
-11 6 0
11 -2 -3 -6 0
-12 4 0
-12 6 0
12 -4 -6 0
x 13 14 0
-13 3 0
-13 4 0
13 -3 -4 0
-14 4 0
-14 5 0
-14 6 0
14 -4 -5 -6 0
```

Fig. 2. CryptoMiniSat input file corresponding to the system of equations in (1)

$K^0_{i,j}$, the AES state matrix; with $0 \leq i, j \leq 3$ where i and j denote the row and column indices, respectively. $K^{r+1}_{i,j}$ denotes the bijective mapping of the user key to the 10 sub-round keys, where $0 \leq r \leq 9$ denotes the number of the rounds. The r^{th} key schedule round is defined by the following transformations:

$$\begin{array}{l} K^{r+1}_{0,0} \leftarrow S(K^r_{1,3}) \oplus K^r_{0,0} \oplus Rcon(r+1) \\ K^{r+1}_{i,0} \leftarrow S(K^r_{(i+1)mod4,3}) \oplus K^r_{i,0}, 1 \leq i \leq 3 \\ K^{r+1}_{i,j} \leftarrow K^{r+1}_{i,j-1} \oplus K^r_{i,j}, 0 \leq i \leq 3, 1 \leq j \leq 3 \end{array} \quad (4)$$

where $Rcon(\cdot)$ denotes a round-dependent constant and $S(\cdot)$ represents the S-box operations based on the 8×8 Rijndael S-box [29]. Figure 3 shows the transformations given by equation 4.

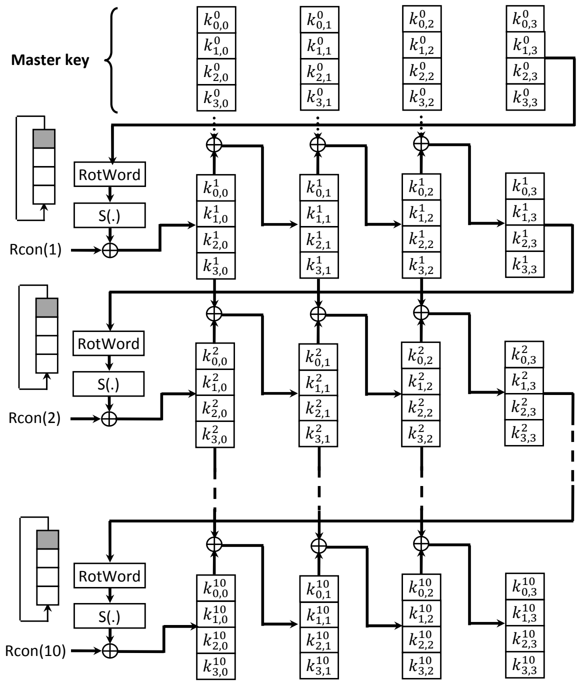

Fig. 3. AES Key Scheduler

4.2 Key Schedule of Serpent

Serpent [31] is a 32 round block cipher based on a substitution permutation network (SPN) structure with an Initial Permutation (IP) and a Final Permutation (FP). It has 32 rounds, each consists of a key mixing operation, a pass through S-boxes, and (in all but the last round) a linear transformation. In the last round, this linear transformation is replaced by an additional key mixing operation. The cipher accepts a variable user key length that is always padded up to 256 bits by appending one bit-value '1' to the end of the most significant bit followed by bit-values '0'. To obtain the 33 128-bit subkeys $K_0, ..., K_{32}$, the user key is divided into eight 32-bit words $w_{-8}, w_{-7}, ..., w_{-1}$, from which the 132 intermediate keys or pre-keys $(w_0 ... w_{131})$ are derived as follows:

$$w_i := (w_{i-8} \oplus w_{i-5} \oplus w_{i-3} \oplus w_{i-1} \oplus \phi \oplus i) <<< 11 \tag{5}$$

where ϕ is a constant formed by the fractional part of the golden ratio $(\sqrt{5}+1)/2$ or 0x9e3779b9 in hexadecimal.

The round keys k_i are evaluated from the pre-keys by first calling one of the eight 4×4 S-boxes in bit slice mode. In bit slice mode, each input of the S-box comes from a different 32-bit word and each output goes to a different 32-bit word. The 4x32 bits per round are all handled by the same S-box. A group of four input or

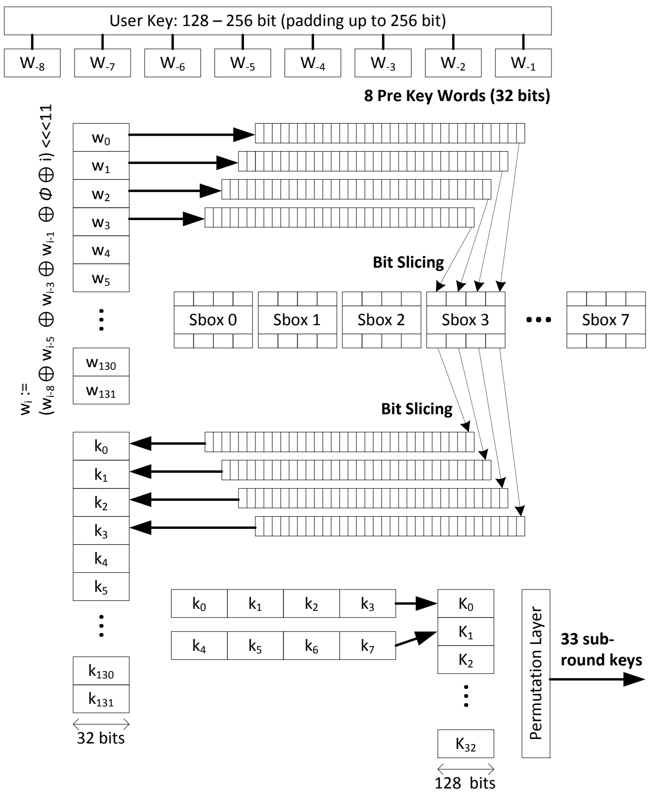
User Key: 128 – 256 bit (padding up to 256 bit)
W-8 W-7 W-6 W-5 W-4 W-3 W-2 W-1
8 Pre Key Words (32 bits)
wi := (wi-8 ⊕ wi-5 ⊕ wi-3 ⊕ wi-1 ⊕ Φ ⊕ i) <<11
w0 w1 w2 w3 w4 w5 w130 w131
Bit Slicing
Sbox 0 Sbox 1 Sbox 2 Sbox 3 Sbox 7
Bit Slicing
k0 k1 k2 k3 k4 k5 k130 k131
32 bits
k0 k1 k2 k3
k4 k5 k6 k7
K0 K1 K2 K32
128 bits
Permutation Layer
33 sub-round keys

Fig. 4. Serpent Key Scheduler

four output words defines a unit that is handled together. The transformation from pre-keys w_i into words k_j of round keys is performed as follows:

$$\begin{array}{rcl}
\{k_0;k_1;k_2;k_3\} & = & S_3(w_0;w_1;w_2;w_3)\\
\{k_4;k_5;k_6;k_7\} & = & S_2(w_4;w_5;w_6;w_7)\\
\{k_8;k_9;k_{10};k_{11}\} & = & S_1(w_8;w_9;w_{10};w_{11})\\
\{k_{12};k_{13};k_{14};k_{15}\} & = & S_0(w_{12};w_{13};w_{14};w_{15})\\
\{k_{16};k_{17};k_{18};k_{19}\} & = & S_7(w_{16};w_{17};w_{18};w_{19})\\
\vdots & \vdots & \vdots\\
\{k_{124};k_{125};k_{126};k_{127}\} & = & S_4(w_{124};w_{125};w_{126};w_{127})\\
\{k_{128};k_{129};k_{130};k_{131}\} & = & S_3(w_{128};w_{129};w_{130};w_{131})
\end{array} \tag{6}$$

where S_i denotes the i^{th} s-box of Serpent. The round keys K_i are then formed by regrouping the 32-bit values k_j as 128-bit sub-keys K_i (for i $\in$ 0,.., r) as follows:

$$K_i := \{k_{4i};k_{4i+1};k_{4i+2};k_{4i+3}\} \tag{7}$$

Finally, we apply IP to the round keys K_i in order to place the key bits in the correct column, i.e., $\hat{K}_i = \mathrm{IP}(K_i)$. Figure 4 depicts the described key scheduler of Serpent.

By exploiting the asymmetric decay of the memory images and the redundancy of key material inherent in the key schedule of both algorithms above, rectifying the faults in the corrupted memory images of the the key schedule is formulated as a Boolean satisfiability problem which can be solved efficiently for relatively large decay factors.

5 Experimental Results

Because of the nature of the cold boot attack, it is realistic to assume that only a corrupted image of the contents of memory is available to the attacker, i.e., a fraction of the memory bits will be flipped. Halderman *et al.* [3] observed that, within a specific memory region, the decay is overwhelmingly asymmetric, i.e., either $0 \to 1$ or $1 \to 0$. When trying to retrieve cryptographic keys, the decay direction for a region can be determined by comparing the number of 0s and 1s since in an uncorrupted key, the expected number of 0s and 1s should approximately be equal.

Similar to the previous work in [3] [6] [32], throughout our experimental results, we assume an asymmetric decay model where bits overwhelmingly decay to their ground state rather than their charged state. Using this model, only the bits that remain in their charged state are useful to the cryptanalyst since one cannot be sure about the original values of the 0 bits, i.e., whether they were originally 0's or decayed 1's. Let β denote the fraction of decayed bits. If the percentage of 0's and 1's in the original key schedule bits is p_z and $1 - p_z$, respectively, then the fraction, f, of key bits that can be assumed to be known by examining the decayed memory of the key schedule is given by

$$f = 1 - (p_z + \beta \times (1 - p_z)) = (1 - p_z) \times (1 - \beta).$$

Since in an uncorrupted key schedule key, we expect the number of 0's and 1's to be approximately equal, i.e., $p_z \approx 1/2$, then we have $f \approx (1 - \beta)/2$.

In our experiments, the input files for the CryptoMiniSAT contained 5,144 and 18,500 clauses for AES and Serpent, respectively. For PolyBoRi, 1,280 equations with 1,728 variables were defined for AES and 8,448 equations with 8,704 variables were defined for Serpent.

Tables 1, 2, 3 [32] and 4 show statistics for the run time required to recover the key of AES and Serpent from the corresponding corrupted memory images for different decay factors. These runtime statistics were obtained using PolyBoRi and CryptoMiniSat running on a Dell Precision 370 workstation with a 3.0 GHz Intel Pentium 4 CPU and 1 GB of RAM. Examining the results in the tables reveal the following observations:

– While the resource requirements of both tools (time for CryptoMiniSat, and time and memory for PloyBoRi) seem to grow exponentially with the decay factor, for practical values of the decay factor, both tools require reasonably short time to recover the secret keys from corrupted memory images.

- The simple and high redundancy in the AES key schedule allows for faster recovery of the key from corrupted memory images. This makes AES more prone to these attacks as compared to other AES finalist such as Serpent. In fact, our initial experiments with Twofish [33] indicate that its relatively more complex key schedule limits the practical applications of these tools to very small values of the decay factor.
- CryptoMiniSat seems to be more suitable for applications in this type of attacks. In particular, every time we tried to push the decay factor higher than the values reported in Table 1, the PolyBoRi tool always crashed after few minutes due to the excessive memory consumption. This behavior also persisted on a 64 bit Linux operating systems with a freshly compiled PolyBoRi/sage system and 8 GB RAM. The question remains if solutions for a higher decay factor can be achieved in a reasonable time if this memory limitations is fixed in the tool.

Table 1. Run-time statistics using Gröbner basis for AES

Decay	10%	20%	30%	40%	50%	60%	70%
Min	0.4	0.6	0.8	1.3	2.2	3.5	7
Max	0.7	0.9	1.1	2.1	3.6	7.6	45
Avg.	0.6	0.7	0.9	1.7	2.9	5.6	21
St.Dev	0.1	0.1	0.1	0.3	0.5	1.2	13
Med.	0.5	0.7	0.9	1.7	2.9	5.3	15

Table 2. Run-time statistics using Gröbner basis for Serpent

Decay	10%	20%	30%	40%	50%	60%	70%
Min	8	9	17	56	114	417	-
Max	9	34	50	2075	2812	578	-
Avg.	8	15	36	328	399	507	-
St.Dev	0.3	7	11	656	764	47	-
Med.	8	12	40	107	131	504	-

Table 3. Run-time statistics using SAT-solver for AES

Decay	30%	40%	50%	60%	70%
Min	0.046	0.046	0.062	0.062	0.078
Max	0.593	0.140	0.187	0.593	207.171
Avg.	0.064	0.066	0.074	0.102	1.233
St.Dev	0.009	0.007	0.008	0.028	4.899
Med.	0.062	0.062	0.078	0.093	0.359

Table 4. Run-time statistics using SAT-solver for Serpent

Decay	10%	20%	30%	40%	50%	60%	70%
Min	0.4	0.4	0.5	0.5	0.7	0.9	4
Max	0.7	0.8	1.2	1.6	8.0	69	35282
Avg.	0.6	0.6	0.7	0.9	1.9	8	1278
St.Dev	0.05	0.07	0.10	0.22	1.30	11	4402
Med.	0.15	0.17	0.20	0.35	1.18	9	27706

6 Conclusion

In this work, we investigated the suitability of two off-the-shelf Algebraic tools for extraction of cryptographic keys from corrupted memory images. Based on our experimental results, it is clear that while the CryptoMiniSat requires a slightly longer preprocessing step to prepare its input file, this step is done only once and the tool runs much faster than the Gröbner basis PolyBoRi tool. Furthermore, CryptoMiniSat does not require a large amount of memory during run time. However, if several solutions were possible for the SAT problem in question, only one result is returned by the solver and the additional solutions have to be explicitly searched again by re-running the tool after appending some extra constraints to exclude already found solutions. On the other hand, Gröbner basis returns a general form representing all possible solutions. However, PolyBoRi requires large memory and usually crashes when the memory requirements is exceeded which limits its applications for solving large problems. It should also be noted that, given the high redundancy of the key schedules of the considered ciphers, the advantage of being able to return all possible solutions does not seem to be very significant since in all the instances we considered, only one possible solution exists.

References

1. Kocher, P.C.: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996)
2. Kocher, P.C., Jaffe, J., Jun, B.: Differential Power Analysis. In: Wiener, M.J. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
3. Halderman, J.A., Schoen, S.D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J.A., Feldman, A.J., Appelbaum, J., Felten, E.W.: Lest We Remember: Cold Boot Attacks on Encryption Keys. In: van Oorschot, P.C. (ed.) USENIX Security Symposium, pp. 45–60. USENIX Association (2008)
4. Heninger, N., Shacham, H.: Reconstructing RSA Private Keys from Random Key Bits. In: Halevi, S. (ed.) CRYPTO 2009. LNCS, vol. 5677, pp. 1–17. Springer, Heidelberg (2009)
5. Skorobogatov, S.: Low temperature data remanence in static RAM. Technical Report UCAM-CL-TR-536, University of Cambridge, Computer Laboratory (2002)
6. Tsow, A.: An Improved Recovery Algorithm for Decayed AES Key Schedule Images. In: Jacobson Jr., M.J., Rijmen, V., Safavi-Naini, R. (eds.) SAC 2009. LNCS, vol. 5867, pp. 215–230. Springer, Heidelberg (2009)
7. Albrecht, M., Cid, C.: Cold Boot Key Recovery by Solving Polynomial Systems with Noise. In: Lopez, J., Tsudik, G. (eds.) ACNS 2011. LNCS, vol. 6715, pp. 57–72. Springer, Heidelberg (2011)
8. CryptoMiniSat, `http://www.msoos.org/cryptominisat2/` (accessed November 2012)
9. Brickenstein, M., Dreyer, A.: PolyBoRi: A framework for Gröbner-basis computations with Boolean polynomials. Journal of Symbolic Computation, 1326–1345 (September 2009)
10. Shamir, A., van Someren, N.: Playing 'Hide and Seek' with Stored Keys. In: Franklin, M. (ed.) FC 1999. LNCS, vol. 1648, pp. 118–124. Springer, Heidelberg (1999)
11. Maartmann-Moe, C., Thorkildsen, S.E., Årnes, A.: The persistence of memory: Forensic identification and extraction of cryptographic keys. Digital Investigation, 132–140 (2009)

12. Kaplan, B.: RAM is Key, Extracting Disk Encryption Keys From Volatile Memory. Master's thesis, Carnegie Mellon University (May 2007)
13. Courtois, N.T., Bard, G.V., Wagner, D.: Algebraic and Slide Attacks on KeeLoq. In: Nyberg, K. (ed.) FSE 2008. LNCS, vol. 5086, pp. 97–115. Springer, Heidelberg (2008)
14. Courtois, N.T., Bard, G.V.: Algebraic Cryptanalysis of the Data Encryption Standard. In: Galbraith, S.D. (ed.) Cryptography and Coding 2007. LNCS, vol. 4887, pp. 152–169. Springer, Heidelberg (2007)
15. Erickson, J., Ding, J., Christensen, C.: Algebraic Cryptanalysis of SMS4: Gröbner Basis Attack and SAT Attack Compared. In: Lee, D., Hong, S. (eds.) ICISC 2009. LNCS, vol. 5984, pp. 73–86. Springer, Heidelberg (2010)
16. Buchberger, B.: Gröbner-Bases: An Algorithmic Method in Polynomial Ideal Theory, ch. 6, pp. 184–232. Reidel Publishing Company, Dodrecht (1985)
17. Buchmann, J., Pyshkin, A., Weinmann, R.-P.: Block Ciphers Sensitive to Gröbner Basis Attacks. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 313–331. Springer, Heidelberg (2006)
18. Eibach, T., Pilz, E., Völkel, G.: Attacking Bivium using SAT solvers. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 63–76. Springer, Heidelberg (2008)
19. Courtois, N.T., O'Neil, S., Quisquater, J.-J.: Practical Algebraic Attacks on the Hitag2 Stream Cipher. In: Samarati, P., Yung, M., Martinelli, F., Ardagna, C.A. (eds.) ISC 2009. LNCS, vol. 5735, pp. 167–176. Springer, Heidelberg (2009)
20. Courtois, N.T., Nohl, K., O'Neil, S.: Algebraic Attacks on the Crypto-1 Stream Cipher in MiFare Classic and Oyster Cards. Cryptology ePrint Archive, Report 2008/166 (2008)
21. Velichkov, V., Rijmen, V., Preneel, B.: Algebraic cryptanalysis of a small-scale version of stream cipher Lex. IET Information Security, 49–61 (June 2010)
22. Mironov, I., Zhang, L.: Applications of SAT Solvers to Cryptanalysis of Hash Functions. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 102–115. Springer, Heidelberg (2006)
23. De, D., Kumarasubramanian, A., Venkatesan, R.: Inversion Attacks on Secure Hash Functions Using SAT Solvers. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 377–382. Springer, Heidelberg (2007)
24. Sugita, M., Kawazoe, M., Perret, L., Imai, H.: Algebraic Cryptanalysis of 58-Round SHA-1. In: Biryukov, A. (ed.) FSE 2007. LNCS, vol. 4593, pp. 349–365. Springer, Heidelberg (2007)
25. Wang, X., Yin, Y.L., Yu, H.: Finding Collisions in the Full SHA-1. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 17–36. Springer, Heidelberg (2005)
26. Segers, A.: Algebraic Attacks from a Groebner Basis Perspective. Master's thesis, Technische Universiteit Eindhoven (October 2004)
27. Boolean Polynomials, Sage Reference Manual V4.7.2, `http://www.sagemath.org/doc/reference/sage/rings/polynomial/pbori.html` (accessed November 2012)
28. Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of the Third Annual ACM Symposium on Theory of Computing, STOC 1971, pp. 151–158. ACM, New York (1971)
29. Daemen, J., Rijmen, V.: The Design of Rijndael: AES - The Advanced Encryption Standard. Springer (2002)
30. Federal Information Processing Standards Publication (FIPS 197). Advanced Encryption Standard, AES (2001)

31. Anderson, R., Biham, E., Knudsen, L.: Serpent: A Proposal for the Advanced Encryption Standard, `http://www.cl.cam.ac.uk/~rja14/serpent.html` (accessed October 2012)
32. Kamal, A., Youssef, A.: Applications of SAT Solvers to AES Key Recovery from Decayed Key Schedule Images. In: 2010 Fourth International Conference on Emerging Security Information Systems and Technologies (SECURWARE), pp. 216–220 (July 2010)
33. Twofish, `http://www.schneier.com/twofish.html` (accessed September 2012)