Journal on Satisfiability, Boolean Modeling and Computation 9 (2014) 1-25

Applications of SAT Solvers in Cryptanalysis: Finding Weak Keys and Preimages

Frédéric Lafitte frederic.lafitte@rma.ac.be
Department of Mathematics
Royal Military Academy
Belgium

Jorge Nakahara Jr jorge.nakahara@ulb.ac.be
Department of Computer Science
Université Libre de Bruxelles (ULB)
Belgium

Dirk Van Heule dirk.van.heule@rma.ac.be
Department of Mathematics
Royal Military Academy
Belgium

Abstract

This paper investigates the power of SAT solvers in cryptanalysis. The contributions are two-fold and are relevant to both theory and practice. First, we introduce an efficient, generic and automated method for generating SAT instances encoding a wide range of cryptographic computations. This method can be used to automate the first step of algebraic attacks, i.e. the generation of a system of algebraic equations. Second, we illustrate the limits of SAT solvers when attacking cryptographic algorithms, with the aim of finding weak keys in block ciphers and preimages in hash functions. SAT solvers allowed us to find, or prove the absence of, weak-key classes under both differential and linear attacks of full-round block ciphers based on the International Data Encryption Algorithm (IDEA), namely, WIDEA-n for $n \in \{4, 8\}$, and MESH-64(8). In summary: (i) we have found several classes of weak keys for WIDEA-n and (ii) proved that a particular class of weak keys does not exist in MESH-64(8). SAT solvers provided answers to two complementary open problems (presented in Fast Software Encryption 2009): the existence and non-existence of such weak keys. Although these problems were supposed to be difficult to answer, SAT solvers provided an efficient solution. We also report on experimental results about the performance of a modern SAT solver as the encoded cryptanalytic tasks become increasingly hard. The tasks correspond to preimage attacks on reduced MD4 algorithm.

KEYWORDS: *SAT solvers, weak keys, preimage attacks, automated cryptanalysis, algebraic cryptanalysis*

Submitted August 2013; revised February 2014; published August 2014

1. Introduction

This paper describes two applications of SAT solvers: finding weak keys in block ciphers and preimages in hash functions. The experimental framework is implemented and made available in the free open-source software cryptosat [10].

What is a weak key? Generically, a weak key of a block cipher is a user key which leads to a nonrandom behavior of the cipher. Ideally, a block cipher should have no weak keys. For IDEA and related ciphers mentioned previously, a weak key causes some multiplicative subkeys to have value 0 or 1, turning multiplication into a linear operation. These multiplicative subkeys are mandatory inputs to a multiplication operation over $\mathrm{GF}(2^{16}+1)$, where $0 \equiv 2^{16}$ by construction [19]. Note that $2^{16}+1$ is a prime number. We have found classes of weak keys of block ciphers based on the International Data Encryption Algorithm (IDEA) [19]. These block ciphers follow the Lai-Massey scheme and include: WIDEA-n [17] for $n \in \{4, 8\}$ and MESH-8 [29].

Consequences of weak keys in a hash function setting. Ciphers that allow weak keys are not ideal cryptographic primitives. Since block ciphers are pervasive building blocks in other cryptographic constructions, the presence of weak keys leads to a nonrandom behavior not only of the encryption/decryption framework, but also of higher-level structures, such as hash functions and stream ciphers. For instance, in a hash function setting, suppose a block cipher E is used as a permutation component in a compression function in Davies-Meyer (DM) mode: $h_i = h_{i-1} \oplus E_{m_i}(h_{i-1})$ where $h_0 = \mathrm{IV}$ is an initial value. In the HIDEA-n hash function [16], for instance, WIDEA-n ciphers are used as permutations E in DM mode. In this setting, h_i and h_{i-1} are chaining variables, and the key K is a message block m_i, which is under the control of the adversary. If K is a weak value, then $E_{m_i} = E_K$'s behavior can be controlled indirectly, to cause an output difference equal to the input difference, i.e. $\Delta E_{m_i}(h_{i-1}) = \Delta h_{i-1}$. In other words, the adversary uses the feedforward of h_{i-1} in the DM mode and the weak key to cause difference $\Delta h_i = h_i \oplus h'_i = 0$, that is, a collision, with certainty. Therefore, the presence of weak keys leads to collisions with only two compression function computations, which is much faster than the expected complexity due to the birthday paradox. Other consequences of weak keys are discussed in the following paragraphs.

Consequences of weak keys in a block cipher setting. Let us consider a pure encryption setting, with an m-bit block and k-bit cipher (where m is a multiple of 16 for the ciphers targeted in this paper). To detect if a key K is weak, one can ask for the encryption of two plaintext blocks (P, P') such that $\Delta P = P \oplus P' = (\delta, \delta, \ldots, \delta)$, where $\delta = 8000_x$. If K is weak according to a differential path, for example, as described in [30] for WIDEA-4, then the corresponding ciphertexts (C, C') will satisfy $C \oplus C' = P \oplus P'$ with certainty. Otherwise, for a random key, this equality will be satisfied with probability 2^{-m}. This test can be implemented with two chosen plaintexts and two encryptions. For example, for WIDEA-4, the given differential

patterns hold with certainty for a weak key but with 2^{-256} chance for a random key. That means that for two plaintext pairs, we expect $2^{512-2\cdot256} = 1$ false alarm. Other classes of weak keys can be detected, depending on the differential pattern used.

Consequences of weak keys in a stream cipher setting. In a stream cipher construction, such as OFB, CTR and CFB modes [25], the occurrence of a weak key in the underlying block cipher means that there is a non-negligible correlation (or bias) in the keystream bits generated by these modes, which can be exploited by a linear cryptanalytic attack [30, 4]). In particular, for IDEA, WIDEA-n and MESH-64(8), the bias is 2^{-1} if a weak keys is used. This bias is the maximum, which means the data complexity is as low as $8 \cdot (2^{-1})^{-2} = 32$ known text blocks. Thus, in a known-plaintext setting, the keystream can be recovered efficiently since the bias is so high. Therefore, the keystream can be distinguished from a random keystream (One-Time Pad).

In summary, the existence of weak keys has a non-negligible security impact because they jeopardize the use of block ciphers in a number of applications, where block ciphers are fundamental building blocks, even if the exact number of weak keys is not clear. The next paragraphs provide background on SAT-based cryptanalysis.

SAT solvers and cryptanalysis. The Boolean satisfiability problem (SAT) was mentioned as a useful framework for cryptanalysis for the first time in [22]; the relation between input and output bits of a cryptographic algorithm can be expressed as a SAT problem which can then be fed to a SAT-solver in order to recover the secret bits (e.g. key bits). The problem SAT can be solved by different types of solvers. For our purposes, we consider only complete SAT solvers, i.e. deterministic algorithms that are given a formula *in Conjunctive Normal Form (CNF)* and that always output a satisfying valuation of its variables, in case such a valuation exists. Otherwise, the SAT solver outputs that the SAT instance is unsatisfiable. A CNF is a conjunction of clauses, a clause is a disjunction of literals, and a literal is a propositional variable or its negation.

SAT solvers and algebraic attacks. SAT-based cryptanalysis is reminiscent of algebraic attacks, since both approaches aim at solving an encoding of the computations underlying a cryptographic algorithm in the secret variables e.g. key bits. In the case of algebraic attacks, the encoding is a system of multivariate equations where variables, over the finite field GF(2) (resp. GF(2^8) or GF(2^{32})), correspond to bits (resp. bytes or words). Then, the system of equations is solved in the secret variables using various algebraic techniques [1]. In the case of SAT-based cryptanalysis, the encoding is a CNF, where the propositional variables correspond to the bits used in the computations. This encoding is then handed to an off-the-shelf SAT solver, instead of an algebraic solver, in order to recover the value of secret variables.

Automating the first step of algebraic attacks. Algebraic attacks are mounted in two phases: (i) generating a system of equations, (ii) solving the system in the secret

variables. It is straightforward to convert a SAT instance into a system of equations over GF(2). For instance, the CNF formula $(x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_4)$ can be written as the system of equations

$$\begin{aligned} 0 &= \mathsf{ANF}(x_1 \vee \neg x_2) \oplus 1 \\ 0 &= \mathsf{ANF}(x_2 \wedge \neg x_4) \oplus 1 \end{aligned} \tag{1}$$

where $\mathsf{ANF}(\cdot)$ denotes a function that returns the Algebraic Normal Form of the input Boolean function. Although the translation from CNF to ANF described above is very poor, it shows that it is possible to obtain automatically a system of algebraic equations relating secret and public bits by generating the CNF first. It is worth mentioning that so far SAT instances were obtained from multivariate equations [15, 2], whereas our method bypasses the algebraic modeling of the target algorithm.

The difficulty of solving the generated instances is shown experimentally by attacking the preimage resistance of the MD4 algorithm. The compression function of MD4 consists of 48 iterations of an unbalanced Feistel ladder. We show how the time and memory required by a modern SAT solver [33] behaves as a function of the number of iterations and the number of unknown preimage bits. The SAT solver was able to invert up to 31 (out of 48) iterations in a few hours using a personal computer (see section 2.2 for details). It was shown in previous research [14] that by taking into account previous cryptanalysis by Dobbertin [6], a SAT solver could invert up to 39 iterations in less than 8 hours.

Outline. This paper is organized as follows: related work is described in Sect. 1.1. Sect. 1.2 presents our contributions. Sect. 2.1 introduces the method used to generate the SAT-instances. Sect. 2.2 presents results on SAT-based preimage attacks on the MD4 algorithm. Sect. 3 presents weak keys for WIDEA-n; Sect. 4 discusses weak keys for the MESH-64(8) cipher. We conclude in Sect. 5.

1.1 Related Work

SAT-based cryptanalysis. Expressing cryptanalysis as a SAT problem was first proposed back in 2000 [22], where SAT solvers were used in an attempt to recover DES keys. In 2005, the authors of [14] proposed a method for generating SAT instances based on C++ operator overloading and used it to compare the hash functions MD4 and MD5 by analyzing their preimage resistance against a SAT solver. Later, SAT solvers were used in [26] to automate parts of the collision attack of Wang *et al.* [35]. In [26], it is mentioned that although SAT solvers alone cannot break modern cryptographic algorithms, they are useful at enhancing cryptanalysis, and their combination to other cryptanalytic techniques seems promising. In [5], the authors show that by taking into account previous cryptanalysis by Dobbertin on MD4 [6], a SAT solver is able to invert 39 iterations (out of 48) of the Feistel ladder used in MD4's compression function, instead of 28 iterations when not using Dobbertin's results.

The cryptanalysis of Dobbertin [6] provided information that could be injected in the CNF encoding of MD4, helping the solver considerably. An adversary might inject other useful information obtained by other means, such as side-channel attacks [27], cold-boot attacks [18], or simple eavesdropping [11]. Other examples of SAT-based cryptanalysis include recovering the internal state of stream ciphers [8, 34, 23], key-recovery algebraic attacks on Keeloq [3], as well as preimage search on hash functions [28]. SAT solvers are often used to assist algebraic attacks in solving systems of equations. A comparison of the performance of SAT solvers with that of algebraic attacks based on Gröbner basis algorithms can be found in [9].

WIDEA-n. In [17], Junod and Macchetti presented a Wide-block version of the IDEA cipher [19] called WIDEA-n, $n \in \{4, 8\}$, combining n instances of the 8.5-round IDEA cipher joined by an $n \times n$ matrix derived from a Maximum Distance Separable (MDS) code placed inside the Multiplication-Addition (MA) box in each round of each IDEA instance. In [30], Nakahara described differential and linear attacks under weak-key assumptions, but no weak keys were presented. In this paper, we explicitly show weak (user) keys that lead to weak subkeys for differential and linear distinguishers across the full 8.5-round WIDEA-n ciphers under the original key schedule algorithms.

Weak keys for WIDEA-8 were investigated by Mendel *et al.* in ([24]), where they described attacks on WIDEA-8 as compression function in Davies-Meyer mode. They used a guess-and-determine approach but only found weak keys for 7.5 rounds, instead of 8.5 rounds. These weak keys allowed them to find free-start collisions for the compression function.

1.2 Our Results

The contributions of this paper include:

- Weak keys of the full 8.5-round WIDEA-4 and WIDEA-8 block ciphers under differential and linear attacks. We are not aware of any such keys ever been reported previously for the full ciphers. The designers of WIDEA-n cipher claimed (in [17], Sect.4.2) that weak keys like in IDEA did not exist in WIDEA-n. Nonetheless, such keys were found efficiently using SAT solvers, for both full 8.5-round ciphers.

- Encoding the search for weak keys of specific differential or linear pattern for the MESH-64(8) cipher resulted in an unsatisfiable SAT instance. This result proves that this particular weak-key class does not exist in MESH-64(8). Therefore, SAT solvers are able to formally rule out a class of attacks.

- An efficient and fully automated method for generating SAT instances encoding a wide range of cryptanalytic tasks. This method is an improvement of previous work [14] and can be easily extended to other cryptographic primitives than the

ones attacked in this paper. It is implemented in the extensible free open-source software cryptosat [10], along with functionalities that allow for the user to easily manipulate and solve instances.

- The first phase of algebraic cryptanalysis, i.e. the generation of a system of equations, can be automated using our SAT-instance generation method. So far, such systems were obtained manually after analysis of the target algorithm. In our case, as shown in equation (1), it is straightforward to automatically transform the SAT instance into a system of sparse multivariate equations over GF(2).

- As mentioned in previous research [26], the combination of SAT solvers with known cryptanalysis is promising. However, examples of such combinations are scarce (see Sect. 1.1 for related work). This paper describes the first application of SAT solvers to find weak keys of block ciphers.

- We also illustrate the limits of SAT solvers by attacking the preimage resistance of MD4's compression function with the SAT solver cryptominisat [33] version 2.9.5 [1]. The solver alone was able to invert up to 31 iterations (out of 48) in a few hours (see section 2.2).

Reproducibility. All results were obtained using the free open-source software cryptosat [10] version 0.1.0 running in the R environment [32] version 3.0.2 using cryptominisat [33] version 2.9.5 and the scripts given in the appendix. All randomness in these scripts comes from the default pseudo-random number generator of R which is the Mersenne-Twister.

2. Generating and Solving SAT Instances

In previous work, [2, 27, 3], the CNF representation of a cryptographic algorithm was obtained by translating a system of algebraic equations into a set of clauses. In this section we present an efficient and fully automated method for generating CNF encodings of a wide range of cryptographic computations. The method outputs the SAT instance directly in DIMACS (a standard format for Boolean functions in CNFs) and has been used for attacking symmetric-key algorithms that use only the bitwise operators NOT ($\neg$), XOR ($\oplus$), AND ($\wedge$), OR ($\vee$), as well as addition modulo 2^n ($\overset{n}{\boxplus}$) and left rotation of amplitude s on n-bit words ($\overset{n}{\lll} s$).

This set of operations is sufficient in order to attack ARX algorithms (i.e. algorithms based on mixing modular addition, rotation and bitwise exclusive or), but other operators, such as multiplication, can be encoded in a similar way.

1. Similar figures were obtained using minisat [7] (version 2.2.0).

We then present the performance of a modern SAT solver, namely cryptominisat [33], in terms of time and memory, when solving increasingly difficult encodings of preimage attacks on the MD4 algorithm.

2.1 Generating SAT Instances

In [14], the authors proposed an elegant method based on C++ operator overloading for generating SAT instances. The C++ code of each operator ($\oplus$,$\vee$,etc.) is enhanced with a functionality that updates a global propositional formula, encoding the cryptographic operations as they take place. However, this approach requires to maintain a (huge) formula in memory, and to translate it into CNF [12].

Our method is an improvement over [14] that consists in expressing each operator directly in CNF so that the overloaded code can output the global formula directly to a file, clause after clause, as the operations are executed. This makes the generation efficient since it requires practically no memory nor additional computations to handle the formula.

For example, let us consider the operation $c = a \vee b$, with $a, b, c \in \{0,1\}^{32}$, and let $f_\vee$ denote the corresponding formula:

$$f_\vee \quad = \quad \bigwedge_{i=1}^{32}(c_i \leftrightarrow (a_i \vee b_i)) \overset{CNF}{=} \bigwedge_{i=1}^{32}(c_i \vee \neg a_i) \wedge (c_i \vee \neg b_i) \wedge (\neg c_i \vee a_i \vee b_i)$$

Each time operator $\vee$ is executed, its overloaded code will generate these 32×3 clauses, with the effective variable identifiers instead of the formal identifiers a_i, b_i, c_i. As the clauses are generated, these variables are simply identified using a counter that is incremented in the overloaded code.

The CNF of the other operators (i.e. $f_\wedge, f_\oplus, f_\neg, f_{\lll s}, f_\boxplus$) are obtained in a similar way (for the $\boxplus$ operator, variables d_i denote the carry bit):

$$f_\vee \overset{\text{CNF}}{=} \bigwedge_{i=1}^{32}(c_i \vee \neg a_i) \wedge (c_i \vee \neg b_i) \wedge (\neg c_i \vee a_i \vee b_i)$$

$$f_\wedge \overset{\text{CNF}}{=} \bigwedge_{i=1}^{32}(c_i \vee \neg a_i \vee \neg b_i) \wedge (\neg c_i \vee a_i) \wedge (\neg c_i \vee b_i)$$

$$f_\oplus \overset{\text{CNF}}{=} \bigwedge_{i=1}^{32}(c_i \vee a_i \vee \neg b_i) \wedge (c_i \vee b_i \vee \neg a_i) \wedge (\neg c_i \vee \neg a_i \vee \neg b_i) \wedge (\neg c_i \vee a_i \vee b_i)$$

$$f_\neg \overset{\text{CNF}}{=} \bigwedge_{i=1}^{32}(c_i \vee a_i) \wedge (\neg c_i \vee \neg a_i)$$

$$f_{\lll s} \overset{\text{CNF}}{=} \bigwedge_{i=1}^{32}(c_i \vee \neg a_{i+s \mod 32}) \wedge (\neg c_i \vee a_{i+s \mod 32})$$

$$
\begin{aligned}
f_{\boxplus} &\overset{\text{CNF}}{=} \neg d_0 \bigwedge_{i=1}^{32} \text{CNF}(\varphi_i) \wedge \text{CNF}(\psi_i) \\
\varphi_i &= (\text{XOR}(a_i, b_i, d_{i-1}) \vee \neg d_i) \wedge (\neg\text{XOR}(a_i, b_i, d_{i-1}) \vee d_i) \\
\psi_i &= (\text{MAJ}(a_i, b_i, d_{i-1}) \vee \neg d_i) \wedge (\neg\text{MAJ}(a_i, b_i, d_{i-1}) \vee d_i)
\end{aligned}
$$

As noted in [14], by using operator overloading, the generation of the clauses is independent of the target cryptographic algorithm. Therefore, the same implementation can easily be applied to any other cryptographic algorithm that uses these operators.

Although the translation described in this section is far from optimal, it does not lead to poor solver performances compared to previous work [5] as shown in the next section. In particular, we could invert 31 steps of MD4 using this flat encoding.

2.2 Solving SAT Instances

This section reports experimental results when applying the SAT solver cryptominisat to preimage attacks on (reduced versions of) the MD4 algorithm. The MD4 algorithm inspired the design of a wide range of cryptographic hash functions, including MD5, SHA-1, and SHA-2 among other hash functions [25]. As mentioned in [20], MD4 is still used in some widespread protocols, most notably the S/KEY one-time password protocol [13] and the NT LAN Manager (NTLM) authentication protocol.

MD4 uses a strengthened Merkle-Damgård mode of operation [25], with a compression function that operates on 32-bit words. The input to the compression function consists of sixteen message words denoted $W_0, ..., W_{15}$, and four words denoted a, b, c, d which constitute the internal state. One call to the compression function updates this internal state by iterating the unbalanced Feistel transformation shown in Fig 1. That is, starting from the initial state IV $= (a_0, b_0, c_0, d_0)$, the transformation in Fig 1 is applied iteratively to yield $(a_{48}, b_{48}, c_{48}, d_{48})$, using one message word in each step, denoted w_i with $i \in \{1, \ldots, 48\}$. After the 48 steps, each word $W_0, ..., W_{15}$ has been used three times according to a pre-specified order. Finally, the compression function outputs the final state $(a_0 \boxplus a_{48}, b_0 \boxplus b_{48}, c_0 \boxplus c_{48}, d_0 \boxplus d_{48})$.

Note that in Fig. 1 each step computes a single 32-bit value, denoted as in [20] by Q_i, with $i \in \{1, ..., 48\}$. The transformation can be written as equation (2), and as equation (3) after writing the initial values $Q_{-3}, Q_0, Q_{-1}, Q_{-2}$ instead of a_0, b_0, c_0, d_0 respectively.

$$
\begin{aligned}
Q_i &= (a_{i-1} \boxplus \Phi_i(b_{i-1}, c_{i-1}, d_{i-1}) \boxplus w_i \boxplus k_i) \lll s_i \quad \forall i \in \{1, ..., 48\} \qquad (2) \\
Q_i &= (Q_{i-4} \boxplus \Phi_i(Q_{i-1}, Q_{i-2}, Q_{i-3}) \boxplus w_i \boxplus k_i) \lll s_i \quad \forall i \in \{1, ..., 48\} \qquad (3)
\end{aligned}
$$

Inverting the compression function consists in solving the system of equations (3), after plugging in values for the final state obtained from the target hash value. It is

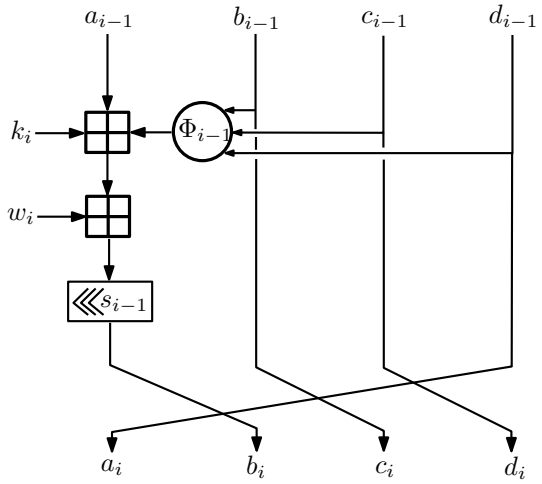

Figure 1. The Feistel ladder of MD4 (all values hold on 32 bits). Each call to the compression function updates the hash function state (a_i, b_i, c_i, d_i) using 48 applications (steps) of this transform. Each step $i \in \{1, ..., 48\}$ combines the message word w_i with the state and step-dependent constant k_i using a step-dependent Boolean function Φ_i, addition modulo 2^{32} ($\boxplus$) and left rotation of amplitude s_i ($\lll s_i$) where s_i is also a step-dependent constant.

Table 1. This 512-bit message hashes to the hexadecimal value $f...f_x$ when applying MD4's compression function reduced to 31 steps.

$W_0 : f460339f_x,$	$W_1 : 52b843bb_x,$	$W_2 : 91032846_x,$	$W_3 : 2ec58249_x,$
$W_4 : 927047df_x,$	$W_5 : 513f1d9f_x,$	$W_6 : 36cea836_x,$	$W_7 : 1179ab4b_x,$
$W_8 : 703e2918_x,$	$W_9 : c220ff73_x,$	$W_{10} : 85db2664_x,$	$W_{11} : 6389fd3c_x,$
$W_{12} : 086cd663_x,$	$W_{13} : c9371d30_x,$	$W_{14} : f77f4fc0_x,$	$W_{15} : 6188439b_x.$

also possible to plug in values of known message bits, which is particularly relevant in the case of a password recovery attack since most of the preimage bits are known to be zero. An attacker might also take advantage of known password characters (e.g. obtained via shoulder surfing).

The method described in section 2.1 allows us to easily generate the SAT instance encoding the system (3). As the number of steps and unknown bits increase, we report the performance (time and memory) required by the solver.

Experimental setup. Let s denote the number of steps and b the number of unknown preimage bits. For each value of $s \in \{15, 20, 22, 24, 25, 26, 27\}$ and $b \in \{100, 200, 300, 400, 500\}$, we generated 250 random DIMACS files by applying the MD4 compression function to 250 random message blocks $x \in \{0, 1\}^{512}$. Then, for each

value of (s, b), we compute the median performance of cryptominisat in terms of CPU time and memory usage.

Experimental results. Figures 2 and 3 illustrate how solving time and memory usage grow as a function of the parameters s and b, using the SAT solver cryptominisat [33] on a personal computer. It is worth noting that the amount of memory required by the SAT solver does not blow up as it is the case with Gröbner basis algorithms [9]. Table 1 shows a preimage that hashes to the hexadecimal value $f...f_x$ after 31 steps of MD4's Feistel transformation. It was found in 17 hours using 1.4 GB of memory with a PC that has 2 GB of memory and a 3.06 GHz CPU with the following characteristics: GenuineIntel i686; 32-bit; little-endian; CPU(s):2; thread(s) per core 1; Core(s) per socket:1.

3. Weak Keys for the WIDEA-n Block Ciphers

Among the several design criteria for key schedule algorithms, one can find: high performance, compact code, low latency, resistance to side-channel analysis, and avoidance of weak keys, among others. The presence of weak keys means a failure of both the key schedule and the encryption/decryption algorithm's designs [25], since the existence of suspicious bit patterns in the key is not a threat if there is no shortcut attack in the encryption/decryption frameworks because of them. Likewise, nonrandom cipher behavior [21] that is unrelated to bit patterns in the key or in the subkeys does not mean the latter are weak. Keys are considered weak only when the apparent weakness propagates from the key schedule to the encryption algorithm, and leads to nonrandom cipher behavior and eventually to (efficient) attacks.

3.1 The Key Schedule of WIDEA-n

Let Z_i, for $0 \leq i \leq 51$, denote the round subkeys used in 8.5-round WIDEA-n, $n \in \{4, 8\}$. The key schedule algorithm of WIDEA-4 is as follows [17]: due to the 4-way parallelism, each subkey has 64 bits. Let K_i, for $0 \leq i \leq 7$, denote the eight 64-bit words representing the user key. The round subkeys are computed by the key schedule as follows:

$$\begin{aligned}
Z_i &= K_i,\ 0 \leq i \leq 7.\\
Z_i &= ((((Z_{i-1} \oplus Z_{i-8}) \overset{16}{\boxplus} Z_{i-5}) \overset{16}{\lll} 5) \lll 24) \oplus C_{i/8-1},\ 8 \leq i \leq 51,\ i \equiv 0 \bmod 8.\\
Z_i &= ((((Z_{i-1} \oplus Z_{i-8}) \overset{16}{\boxplus} Z_{i-5}) \overset{16}{\lll} 5) \lll 24),\ 8 \leq i \leq 51,\ i \not\equiv 0 \bmod 8. \qquad (4)
\end{aligned}$$

where operations superscripted with '16' indicate that the operation is actually carried out over 16-bit slices of Z_i. Otherwise, the operation is over 64-bit words, such as the bitwise left-rotation $\lll$ 24. Following [17], $C_0 = 1dea000000000000_x$, $C_1 = 3825000000000000_x$, $C_2 = 1dd7000000000000_x$, $C_3 = 3ea4000000000000_x$, $C_4 = e57a000000000000_x$, $C_5 = f7a000000000000_x$ are constants inserted every eight

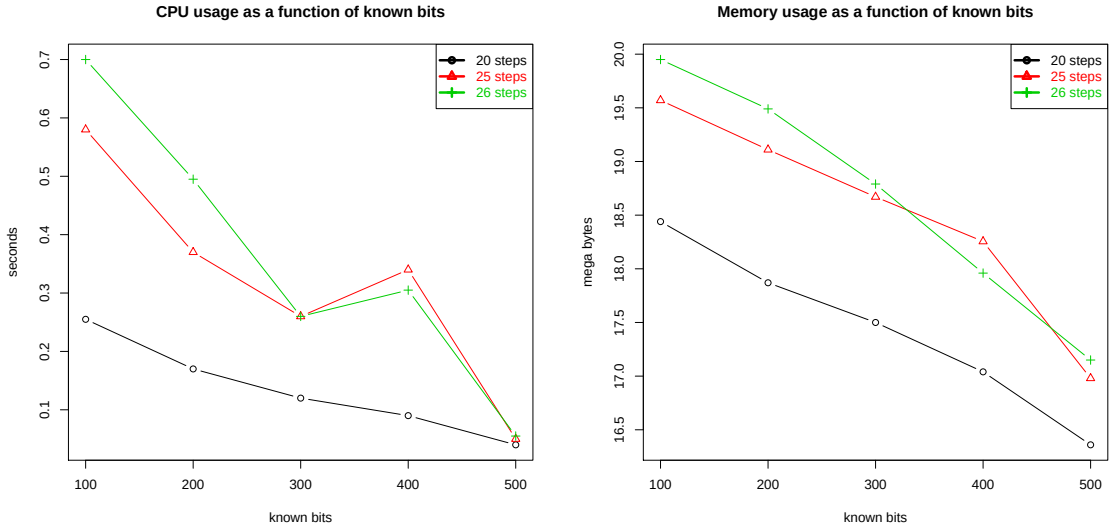

Figure 2. The median CPU time (seconds) and memory (MB) used by the SAT solver over 250 random instances, as a function of the number of unknown bits b. Each curve corresponds to a different value for parameter s.

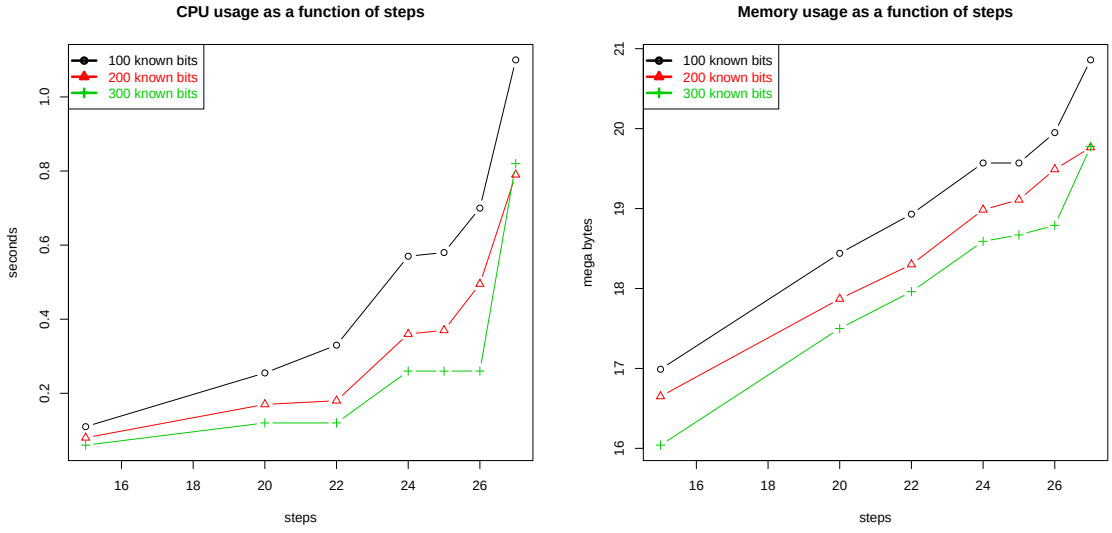

Figure 3. The median CPU time (seconds) and memory (MB) used by the SAT solver over 250 random instances, as a function of the number of steps s. Each curve corresponds to a different choice of parameter b.

rounds. The design of this key schedule uses nonlinear feedback shift registers and was inspired by the key schedule of the MESH ciphers [31].

In total, the key schedule of WIDEA-4 generates 52 subkeys, but the first eight subkeys come from the user key. Each equation in (4) means an equality on 64 bits, and each subkey is also 64 bits long. Therefore, bitwise, there are $(52-8)*64 = 2816$ nonlinear equations in 512 unknowns (key bits). It is an overdefined system of nonlinear equations.

The key schedule algorithm of WIDEA-8 [17] follows (4) in an 8-way-fold parallelism. Let K_i, for $0 \leq i \leq 7$, denote the eight 128-bit words representing the user key. The 128-bit round subkeys are computed exactly as for WIDEA-4. The WIDEA-4 constants $C_{i/8-1}$ are left shifted by 64 bits (padded on the right by 64 zero bits). In total, the key schedule of WIDEA-8 can be seen as a system of 5632 nonlinear equations in 1024 unknowns (keys), double the size of the system for WIDEA-4. Both systems have the same ratio of number of equations (denoted E) over number of variables (V): $E/V = 2816/512 = 5632/1024 = 5.5$.

Let us define an order for each IDEA instance in WIDEA-n: the one whose subkeys are in the most significant 64-bit position will be referred to as first instance, and so forth until the n-th IDEA instance.

The state of WIDEA-4 can be represented by a 4×4 matrix of 16-bit words. Each row of the state corresponds to an IDEA instance.

The best differential and linear distinguishers for the full 8.5-round WIDEA-4, under weak-key assumptions, presented in [30] are 1-round iterative and have the form

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \delta & \delta & \delta & \delta \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \delta & \delta & \delta & \delta \end{pmatrix} \tag{5}$$

in other words, the attack focuses on a single IDEA instance (out of four) in order to minimize the number of weak-subkey assumptions. In a differential cryptanalysis setting, $\delta = 8000_x$ is a 16-bit xor difference. In linear cryptanalysis, $\delta = 1$ is a 16-bit mask.

Suppose a differential setting. For a weak key, the 1-round iterative distinguisher (5) holds with certainty. But, for a random, non-weak key, (5) holds with probability $(2^{-16})^2$ per round. Therefore, weak keys allows to distinguish the full 8.5-round WIDEA-4 with certainty, that is a single plaintext pair, which means a complexity of two chosen plaintexts and equivalent encryption effort. For a random key, (5) repeated 8.5 times would hold with (negligible) probability $(2^{-32})^9 = 2^{-288}$. This is a concrete example of a nonrandom behavior caused by weak keys.

The corresponding weak subkeys consist of 64-bit values whose fifteen most significant bits (MSBs) are zero. In WIDEA-4, this means that the following eighteen subkeys should be weak in their most significant 16 bits: Z_0, Z_3, Z_6, Z_9, Z_{12}, Z_{15}, Z_{18}, Z_{21}, Z_{24}, Z_{27}, Z_{30}, Z_{33}, Z_{36}, Z_{39}, Z_{42}, Z_{45}, Z_{48} and Z_{51}.

For attacking different IDEA instances in WIDEA-4, the pattern of fifteen zero bits must move to other non-overlapping 16-bit pieces of the 64-bit subkey Z_i.

3.2 Results for WIDEA-n

In this section we present examples of weak 512-bit keys for WIDEA-4, found by a SAT solver, for each of the four IDEA instances. Values in bold indicate the weak subkey bits. Subscript x denotes hexadecimal value.

The top part of Table 2 shows a weak key for the first IDEA instance in WIDEA-4, while the bottom half shows the round subkeys. The 15-bit weak subkey piece is highlighted in boldface type. Tables 5, 6 and 7 show weak keys for the other IDEA instances in WIDEA-4.

Table 2. 512-bit user key of WIDEA-4 and weak subkeys concerning the first IDEA instance.

$Z_0 : \mathbf{0000}2d289ea5066c_x,$	$Z_1 : e2e9bcf7b5ab6391_x,$	$Z_2 : 17ed8810ef6ce89a_x,$
$Z_3 : \mathbf{0000}7e1271ac8d08_x,$	$Z_4 : 377cd37c92ae7c46_x,$	$Z_5 : a20c6abecdeda9dc_x,$
$Z_6 : \mathbf{0000}ea1281c20742_x,$	$Z_7 : a7c289d06d0d3e3a_x.$	
$Z_9 : \mathbf{0000}2086d664be7b_x,$	$Z_{12} : \mathbf{0000}4f4fddbb0e63_x,$	$Z_{15} : \mathbf{0000}f17468e835e3_x,$
$Z_{18} : \mathbf{0000}6873bbf1985a_x,$	$Z_{21} : \mathbf{0000}bee7d00e3ce6_x,$	$Z_{24} : \mathbf{0000}2caf3fd879d6_x,$
$Z_{27} : \mathbf{0000}7569bfcbe27a_x,$	$Z_{30} : \mathbf{0000}bab68c0b85b5_x,$	$Z_{33} : \mathbf{0000}fc2b60bcbbef_x,$
$Z_{36} : \mathbf{0000}dcfe48b90bcb_x,$	$Z_{39} : \mathbf{0000}f310e660aaaa_x,$	$Z_{42} : \mathbf{0000}471d7f2aabc8_x,$
$Z_{45} : \mathbf{0000}d37c552d9c9c_x,$	$Z_{48} : \mathbf{0000}e885c4f8bc76_x,$	$Z_{51} : \mathbf{0000}2a3dba933a79_x.$

The given weak keys are just examples, but they already contradict the claims of [17] that no weak keys existed for WIDEA-4. Finding these weak keys took only a few seconds of a SAT solver. Thousands of other weak keys were found just as fast. Therefore, although the key schedule of WIDEA-4 in (4) is more involved than IDEA's, weak keys still exist in both ciphers. Note that the weak key in Table 2 allows to attack only the first IDEA instance in WIDEA-4. Therefore, this weak key cannot be used to attack any of the other three IDEA instances, and vice-versa. Consequently, each weak key belongs to a different, non-overlapping weak-key class. The non-overlapping nature of these classes guarantees that the size of their union is maximal (compared to the overlapping case).

We also searched for weak keys in all four IDEA instances simultaneously in WIDEA-4. The corresponding SAT instance was unsatisfiable, therefore, proving that no such weak keys exist.

The estimate of the weak-key class size in [30] is based on the assumption that each weak subkey requires fifteen bits to be zero. Across 8.5 rounds, with two weak subkeys per round, and attacking two IDEA instances at once, means $9 \cdot 2 \cdot 2 \cdot 15 = 540$ bits restricted to zero (if each subkey condition holds independently). But, this means a probability of 2^{-540}, which for a 512-bit key, means no weak key is expected to exist.

Next, we present examples of weak 1024-bit keys for WIDEA-8, found by a SAT solver, for each of the eight IDEA instances. Therefore, each such weak key belongs to a different weak-key class.

The state of WIDEA-8 can represented by a 8×4 matrix of 16-bit words. Each row corresponds to an IDEA instance. The best differential and linear distinguishers for the full 8.5-round WIDEA-8, under weak-key assumptions, presented in [30] are 1-round iterative and have the form

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \delta & \delta & \delta & \delta \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \delta & \delta & \delta & \delta \end{pmatrix} \tag{6}$$

in other words, the attack focus on a single IDEA instance at a time (out of eight). In a differential cryptanalysis setting, $\delta = 8000_x$ is a 16-bit xor difference. In linear cryptanalysis, $\delta = 1$ is a 16-bit mask. The corresponding weak subkeys consist of 128-bit values whose fifteen most significant bits (MSBs) are zero. In WIDEA-8, this means that the same eighteen subkeys as in WIDEA-4 should be weak.

The top part of Table 3 shows a weak key for the first IDEA instance in WIDEA-8, while the bottom half shows the round subkeys. The 15-bit weak subkey piece is in boldface type. Tables 8 to 14 (in the appendix) show weak keys for the other IDEA instances in WIDEA-8.

Table 3. 1024-bit user key of WIDEA-8 and weak subkeys concerning the first IDEA instance.

$Z_0 : \mathbf{0001}c9c63160bceeac68c62c4d16be85_x,$	$Z_1 : aea162405f37f9d07babc9d3f32a531d_x,$
$Z_2 : ba122e73a3da189d4879b0e05af6d08b_x,$	$Z_3 : \mathbf{0000}aee346b6a872f3b9a7f1f9b3f5d4_x,$
$Z_4 : bdefc6e9fe7c328113798a7b84bd93a3_x,$	$Z_5 : 09569d8d3b26ffd6fbfbbf5186794b24_x,$
$Z_6 : \mathbf{0001}41b5ef0c87fdca638e8c32d7231b_x,$	$Z_7 : fc2f60e69dc434a23666d040470117ac_x.$
$Z_9 : \mathbf{0001}735387097c70e2ad98f803c1f268_x,$	$Z_{12} : \mathbf{0001}c0cf121ebe551e37cc5e343df4cc_x,$
$Z_{15} : \mathbf{0001}bf1718de290b722dc61fc9c73852_x,$	$Z_{18} : \mathbf{0001}4b8c62faab56adcba849880bdc5a_x,$
$Z_{21} : \mathbf{0001}3dd5e3acdb23bf6c553bddf98c2f_x,$	$Z_{24} : \mathbf{0000}4291bed30f6867003c59dba003ff_x,$
$Z_{27} : \mathbf{0001}f986b4abffe9016fc6a91bbf8603_x,$	$Z_{30} : \mathbf{0001}9e310c7a27fdc67d3854325ffc99_x,$
$Z_{33} : \mathbf{0000}81bb2ea6ee8fdcc8bf4d2057fef7_x,$	$Z_{36} : \mathbf{0001}9c4b08cc2785b5f14767b0ae7cf1_x,$
$Z_{39} : \mathbf{0000}e0e35c2505baff6510fefb155e0a_x,$	$Z_{42} : \mathbf{0000}a7b12bbfafdef197ca0f03acb055_x,$
$Z_{45} : \mathbf{0000}7e7af512af2be5fe6093f34142e9_x,$	$Z_{48} : \mathbf{0000}15b14f4fbbff5542104f9046818e_x,$
$Z_{51} : \mathbf{0001}ca4b0125f706f0b65baaca691528_x.$	

4. Weak Keys of MESH-64(8) Ciphers

The design of the MESH-64(8) block cipher [29] followed closely the Lai-Massey scheme used in the IDEA cipher [19]. As far as we are aware of, no weak key has ever been reported for any number of rounds of MESH-64(8).

4.1 Key Schedule of MESH-64(8) Ciphers

Let the 128-bit user key of MESH-64(8) be $K = (K_0, K_1, \ldots, K_{15})$, where $K_i \in \mathbb{Z}_2^8$. The key schedule of MESH-64(8) is as follows: the first sixteen round subkeys Z_i are $Z_i = K_i \oplus c_i$, where $c_0 = 1$ and $c_i = 2 \cdot c_{i-1}$, for $i > 1$, with multiplication in GF(2^8) = GF(2)[x]/$q(x)$, where $q(x) = x^8 + x^4 + x^3 + x + 1$ and '2' is represented by the polynomial 'x' in GF(2^8). The remaining subkeys are computed as follows, where the subscripts are computed modulo 10: $Z_j = (((Z_{j-16} \boxplus Z_{j-12}) \oplus Z_{j-3}) \boxplus Z_{j-2}) \lll 1 \oplus c_j$, where $\lll$ means left bit rotation, and $16 < j \leq 87$.

4.2 Results for MESH-64(8) Ciphers

The weak-key class for MESH-64(8) which we looked for consists of keys for which the following 1-round pattern

$$(\beta, \beta, \beta, \beta, \beta, \beta, \beta, \beta) \rightarrow (\beta, \beta, \beta, \beta, \beta, \beta, \beta, \beta) \tag{7}$$

holds with maximum probability or maximum bias. In a differential cryptanalysis setting, $\beta = 80_x$ is a byte xor difference, while in a linear cryptanalysis setting, $\beta = 1$ is a linear bit mask. The choice of (7) is because (i) it is iterative, and (ii) the difference pattern of only β's minimizes the number of weak-subkey conditions, i.e. minimizes the number of multiplicative subkeys that need to be weak.

The SAT instance that encodes the search of the given weak-key class for MESH-64(8) is not satisfiable. This allows us to conclude on the non-existence of such weak keys in this particular key class only.

5. Conclusions

This paper provided weak 512-bit keys of WIDEA-4 and 1024-bit keys of WIDEA-8 [17]. These weak user keys lead to 1-round iterative differential and linear distinguishers holding with certainty across the full 8.5-round versions of these ciphers. These findings prove that WIDEA-n, $n \in \{4, 8\}$, are not ideal cryptographic primitives. These findings imply attacks in a block/stream cipher and hash function settings, for both distinguish-from-random and key-recovery attacks.

We also showed that SAT solvers are able to prove the nonexistence of a specific weak-key class (7), as it was the case with MESH-64(8) (see Sect. 4.2).

We also described an efficient and automated method for generating SAT instances encoding a wide range of symmetric-key algorithms e.g. ARX algorithms. The method can be extended to other operators in a straightforward way. This

method also allows to automate the generation of multivariate equations over GF(2), thereby facilitating algebraic cryptanalysis [1].

Table 4 summarizes the results in this paper.

Table 4. Summary of weak-key search results for different ciphers.

Cipher	Weak-key Classes	Block size (bits)	Key size (bits)	Rounds	Comments
WIDEA-4	Tables 2–7	256	512	8.5	one weak key per slice
WIDEA-8	Tables 3–14	512	1024	8.5	one weak key per slice
MESH-64(8)	—	64	128	8.5	no such weak key (Sect. 4.2)

References

[1] Gregory V. Bard. *Algebraic Cryptanalysis*. Springer US, 2009.

[2] Gregory V. Bard, Nicolas T. Courtois, and Chris Jefferson. Efficient methods for conversion and solution of sparse systems of low-degree multivariate polynomials over gf(2) via sat-solvers. Cryptology ePrint Archive, Report 2007/024, 2007. http://eprint.iacr.org/.

[3] NicolasT. Courtois, GregoryV. Bard, and David Wagner. Algebraic and slide attacks on keeloq. In Kaisa Nyberg, editor, *Fast Software Encryption*, **5086** of *Lecture Notes in Computer Science*, pages 97–115. Springer Berlin Heidelberg, 2008.

[4] Joan Daemen, René Govaerts, and Joos Vandewalle. Weak keys for idea. In DouglasR. Stinson, editor, *Advances in Cryptology – CRYPTO '93*, **773** of *Lecture Notes in Computer Science*, pages 224–231. Springer Berlin Heidelberg, 1994.

[5] Debapratim De, Abishek Kumarasubramanian, and Ramarathnam Venkatesan. Inversion attacks on secure hash functions using sat solvers. In João Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing - SAT 2007*, **4501** of *Lecture Notes in Computer Science*, pages 377–382. Springer Berlin Heidelberg, 2007.

[6] Hans Dobbertin. Cryptanalysis of md4. *Journal of Cryptology*, **11**(4):253–271, 1998.

[7] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, **2919** of *Lecture Notes in Computer Science*, pages 502–518. Springer Berlin Heidelberg, 2004.

[8] Tobias Eibach, Enrico Pilz, and Gunnar Völkel. Attacking bivium using sat solvers. In Hans Kleine Bning and Xishun Zhao, editors, *Theory and Applications of Satisfiability Testing – SAT 2008*, **4996** of *Lecture Notes in Computer Science*, pages 63–76. Springer Berlin Heidelberg, 2008.

[9] Jeremy Erickson, Jintai Ding, and Chris Christensen. Algebraic cryptanalysis of sms4: Gröbner basis attack and sat attack compared. In Donghoon Lee and Seokhie Hong, editors, *Information, Security and Cryptology – ICISC 2009*, **5984** of *Lecture Notes in Computer Science*, pages 73–86. Springer Berlin Heidelberg, 2010.

[10] Frédéric Lafitte. *cryptosat: SAT-based attacks on cryptographic algorithms*, 2014. R package version 0.1.0.

[11] Philippe Golle and David Wagner. Cryptanalysis of a cognitive authentication scheme (extended abstract). In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, pages 66–70, Washington, DC, USA, 2007. IEEE Computer Society.

[12] Matthew Gwynne. Attacking aes via sat. BSc Dissertation (Swansea), 2010.

[13] Neil Haller. The s/key one-time password system. RFC 1760 (Informational), February 1995.

[14] Dejan Jovanović and Predrag Janičić. Logical analysis of hash functions. In Bernhard Gramlich, editor, *Frontiers of Combining Systems*, **3717** of *Lecture Notes in Computer Science*, pages 200–215. Springer Berlin Heidelberg, 2005.

[15] Philipp Jovanovic and Martin Kreuzer. Algebraic attacks using sat-solvers. *Groups Complexity Cryptology*, **2**(2):247–259, 2010.

[16] Pascal Junod. Idea: Past, present and future. Early Symmetric Crypto (ESC) seminar, January 2010.

[17] Pascal Junod and Marco Macchetti. Revisiting the idea philosophy. In Orr Dunkelman, editor, *Fast Software Encryption*, **5665** of *Lecture Notes in Computer Science*, pages 277–295. Springer Berlin Heidelberg, 2009.

[18] Abdel Alim Kamal and Amr M. Youssef. Applications of sat solvers to aes key recovery from decayed key schedule images. In *Fourth International Conference on Emerging Security Information Systems and Technologies (SECURWARE)*, pages 216–220. CPS, July 2010.

[19] Xuejia Lai, JamesL. Massey, and Sean Murphy. Markov ciphers and differential cryptanalysis. In DonaldW. Davies, editor, *Advances in Cryptology – EUROCRYPT '91*, **547** of *Lecture Notes in Computer Science*, pages 17–38. Springer Berlin Heidelberg, 1991.

[20] Gaëtan Leurent. Md4 is not one-way. In Kaisa Nyberg, editor, *Fast Software Encryption*, **5086** of *Lecture Notes in Computer Science*, pages 412–428. Springer Berlin Heidelberg, 2008.

[21] Gaëtan Leurent. Cryptanalysis of widea. In Shiho Moriai, editor, *Fast Software Encryption*, Lecture Notes in Computer Science, pages 39–51. Springer Berlin Heidelberg, 2014.

[22] Fabio Massacci and Laura Marraro. Logical cryptanalysis as a sat problem. *J. Autom. Reason.*, **24**(1-2):165–203, February 2000.

[23] Cameron McDonald, Chris Charnes, and Josef Pieprzyk. An algebraic analysis of trivium ciphers based on the boolean satisfiability problem. Cryptology ePrint Archive, Report 2007/129, 2007. http://eprint.iacr.org/.

[24] Florian Mendel, Vincent Rijmen, Deniz Toz, and Kerem Varici. Collisions for the widea-8 compression function. In Ed Dawson, editor, *Topics in Cryptology – CT-RSA 2013*, **7779** of *Lecture Notes in Computer Science*, pages 162–173. Springer Berlin Heidelberg, 2013.

[25] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. Discrete Mathematics and Its Applications. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, October 1996.

[26] Ilya Mironov and Lintao Zhang. Applications of sat solvers to cryptanalysis of hash functions. In Armin Biere and CarlaP. Gomes, editors, *Theory and Applications of Satisfiability Testing – SAT 2006*, **4121** of *Lecture Notes in Computer Science*, pages 102–115. Springer Berlin Heidelberg, 2006.

[27] 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.

[28] Paweł Morawiecki and Marian Srebrny. A sat-based preimage analysis of reduced keccak hash functions. *Information Processing Letters*, **113**(10–11):392–397, 2013.

[29] Jorge Nakahara Jr. Faster variants of the mesh block ciphers. In Anne Canteaut and Kapaleeswaran Viswanathan, editors, *Progress in Cryptology - INDOCRYPT 2004*, **3348** of *Lecture Notes in Computer Science*, pages 162–174. Springer Berlin Heidelberg, 2005.

[30] Jorge Nakahara Jr. Differential and linear attacks on the full widea-n block ciphers (under weak keys). In Josef Pieprzyk, Ahmad-Reza Sadeghi, and Mark Manulis, editors, *Cryptology and Network Security*, **7712** of *Lecture Notes in Computer Science*, pages 56–71. Springer Berlin Heidelberg, 2012.

[31] Jorge Nakahara Jr., Vincent Rijmen, Bart Preneel, and Joos Vandewalle. The mesh block ciphers. In Ki-Joon Chae and Moti Yung, editors, *Information Security Applications*, **2908** of *Lecture Notes in Computer Science*, pages 458–473. Springer Berlin Heidelberg, 2004.

[32] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013.

[33] Mate Soos. Cryptominisat 2.5.0, July 2010.

[34] 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*, **5584** of *Lecture Notes in Computer Science*, pages 244–257. Springer Berlin Heidelberg, 2009.

[35] Xiaoyun Wang and Hongbo Yu. How to break md5 and other hash functions. In Ronald Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, **3494** of *Lecture Notes in Computer Science*, pages 19–35. Springer Berlin Heidelberg, 2005.

Appendix A. Generated instances

Algorithm	Iterations/steps	Variables	Clauses
MD4	5	2448	6795
MD4	6	2835	8334
MD4	7	3222	9873
MD4	8	3609	11412
MD4	9	3996	12951
MD4	10	4383	14490
MD4	11	4770	16029
MD4	12	5157	17568
MD4	13	5544	19107
MD4	14	5931	20646
MD4	15	6318	22185
MD4	16	6705	23724
MD4	17	7156	25455
MD4	18	7607	27186
MD4	19	8058	28917
MD4	20	8509	30648
MD4	21	8960	32379
MD4	22	9411	34110
MD4	23	9862	35841
MD4	24	10313	37572
MD4	25	10764	39303
MD4	26	11215	41034
MD4	27	11666	42765
MD4	28	12117	44496
MD4	29	12568	46227
Widea-4	52	20497	71038
Widea-8	52	40897	141614
Mesh-64(8)	88	5009	21416

Appendix B. Scripts for cryptosat

B.1 Weak keys

```
library(cryptosat)


FILENAME <- "JSAT.weakkeys.results.rda"


names <- c("TARGET","SAT","CPU", "MEM", "VARS", "CLAUSES", "ITERATIONS", "WINSTANCE",
           "SLICE", "COMMENTS")
RESULTS <- data.frame(matrix( ncol=length(names), nrow=1))
names(RESULTS) <- names



find.weak.keys <- function(RESULTS, targetname, winstance=NA, slice=NA)
{
  if(!is.na(winstance) && targetname!="wideaks")
    stop("defined␣'instance'␣but␣target␣is␣not␣wideaks")
  if(is.na(winstance) && targetname=="wideaks") {
    warning("using␣default␣instance␣of␣Widea␣(i.e.␣n=4)")
    winstance <- 4
  }
  if(is.na(slice) && targetname=="wideaks")
    stop("must␣specify␣which␣slice␣to␣attack")
  if(targetname=="wideaks" && slice >= winstance)
    stop("bad␣argument␣'slice'␣or␣'winstance'")

  constrained.subkeys <- seq(from=0,to=51,by=3) # for Widea-n (n=={instance})
  if(targetname=="mesh64ks")
    constrained.subkeys <- c( 10, 15, 21, 22, 28, 33, 39, 40, 46, 51 )
  if(targetname=="mesh8ks")
    constrained.subkeys <- c( 0,2,5,7,11,13,14,16,20,22,25,27,31,33,34,40,42,45,47,51,53,
                54,56,60,62,65,67,71,73,74,76,80,82,85,87)

  ### G E N E R A T E   I N S T A N C E
  target <- Target(targetname)
  if(targetname=="wideaks") target$setParameter("instance", winstance)
  instance <- target$generateInstance()

  ### A D D  W E A K  K E Y  C O N S T R A I N T S
  for( i in constrained.subkeys ) {
    varname <- paste("Z",i,sep="")
    varidx <- NA
    if(targetname=="wideaks") {
      varname <- paste(varname,"[",slice,"]",sep="")
      varidx  <- instance$getIdxOf(varname)[1:15]
    }
    if(targetname=="mesh8ks") {
      varidx  <- instance$getIdxOf(varname)
      if(!is.null(dim(varidx))) varidx <- varidx[1:8,varname]
      instance$setVariable(varidx[1], TRUE)
      varidx <- varidx[-1]
    }
    for(idx in varidx)
      instance$setVariable(idx,FALSE)
  }

  ### S O L V E  I N S T A N C E
  solution <- instance$solveWith()
  result <- c( targetname,                    # "TARGET"
               solution$isSAT(),              # "SAT"
```

```
                solution$getSolvingTime(),      # "CPU"
                solution$getMemoryUsage(),      # "MEM"
                instance$getNumVariables(),     # "VARS"
                instance$getNumClauses(),       # "CLAUSES"
                "full",                         # "ITERATIONS"
                winstance,                      # "WINSTANCE"
                slice,                          # "SLICE"
                Sys.info()["nodename"] )        # "COMMENTS"
  RESULTS <- rbind(RESULTS,result)
  save(RESULTS,file=FILENAME)
  RESULTS
}
```

```
t <- "wideaks"
RESULTS <- find.weak.keys(RESULTS, targetname=t, winstance=4, slice=0)
RESULTS <- find.weak.keys(RESULTS, targetname=t, winstance=4, slice=1)
RESULTS <- find.weak.keys(RESULTS, targetname=t, winstance=4, slice=2)
RESULTS <- find.weak.keys(RESULTS, targetname=t, winstance=4, slice=3)
RESULTS <- find.weak.keys(RESULTS, targetname=t, winstance=8, slice=0)
RESULTS <- find.weak.keys(RESULTS, targetname=t, winstance=8, slice=1)
RESULTS <- find.weak.keys(RESULTS, targetname=t, winstance=8, slice=2)
RESULTS <- find.weak.keys(RESULTS, targetname=t, winstance=8, slice=3)
RESULTS <- find.weak.keys(RESULTS, targetname=t, winstance=8, slice=4)
RESULTS <- find.weak.keys(RESULTS, targetname=t, winstance=8, slice=5)
RESULTS <- find.weak.keys(RESULTS, targetname=t, winstance=8, slice=6)
RESULTS <- find.weak.keys(RESULTS, targetname=t, winstance=8, slice=7)
RESULTS <- find.weak.keys(RESULTS, targetname="mesh8ks")
```

B.2 Preimages

```
library(cryptosat)


FILENAME <- "JSAT.preimages.rndKBfalse.rda"
RNDKNOWNBITS <- FALSE


names <- c("SAT","CPU", "MEM", "VARS", "CLAUSES", "STEPS", "KNOWNBITS",
           "RNDKNOWNBITS", "COMMENTS")
RESULTS <- data.frame(matrix( ncol=length(names), nrow=1))
names(RESULTS) <- names


run.with <- function(RESULTS, steps, knownbits, samples) {
  for( sample in 1:samples ) {
    print(paste(sample,"/",samples))

    # G E N E R A T E   T H E   I N S T A N C E
    target <- Target("md4compress")
    target$setParameter("iterations", steps)
    instance <- target$generateInstance()

    # S E T   T H E   H A S H   V A L U E
    for(i in 1:4) {
      var <- paste("Q[",steps-i,"]",sep="")
      val <- instance$getValueOf(var)
      idx <- instance$getIdxOf(var)
      if(length(val)!=32 || length(idx)!=32) stop("something␣is␣wrong")
```

```
      for(i in 1:length(val))
        instance$setVariable(idx[i],val[i]==1)
    }

    # S E T   T H E   H E L P   B I T S
    Wval <- c() # the known message
    Widx <- c() # corresponding propositional variable indices
    for(i in 0:15) {
      Wvar <- paste("W[",i,"]",sep="")
      Wval <- c(Wval, instance$getValueOf(Wvar))
      Widx <- c(Widx, instance$getIdxOf(Wvar))
    }
    if(length(Wval)!=512 || length(Widx)!=512) stop("something␣is␣wrong")
    MSGBITS <- sample(1:512, knownbits)
    if(!RNDKNOWNBITS) MSGBITS <- 1:knownbits
    for( i in MSGBITS )
      instance$setVariable(Widx[i], Wval[i]==1)

    # S O L V E  T H E  I N S T A N C E
    print("solving...")
    solution <- instance$solveWith()
    print("done!")
    result <- c( solution$isSAT(),                # "SAT"
                 solution$getSolvingTime(),       # "CPU"
                 solution$getMemoryUsage(),       # "MEM"
                 instance$getNumVariables(),      # "VARS"
                 instance$getNumClauses(),        # "CLAUSES"
                 steps,                           # "STEPS"
                 knownbits,                       # "KNOWNBITS"
                 RNDKNOWNBITS,                    # "RNDKNOWNBITS"
                 Sys.info()["nodename"] )         # "COMMENTS"
    RESULTS <- rbind(RESULTS,result)
    save(RESULTS, file=FILENAME)
  }
  RESULTS
}



SAMPLES <- 250
KNOWNBITS <- c(100,200,300,400,500)
STEPS <- c(15,20,22,24,25,26,27)
count <- 0
total <- SAMPLES*length(KNOWNBITS)*length(STEPS)

for(knownbits in KNOWNBITS)
  for(steps in STEPS) {
    print(paste("******␣Steps␣=",steps,"Known␣bits␣=",knownbits,"******"))
    RESULTS <- run.with(RESULTS, steps, knownbits, samples=SAMPLES)
    count <- count + SAMPLES
    print(paste(count,"/",total))
  }
```

Appendix C. Weak keys of WIDEA-n ciphers

Table 5. 512-bit user key of WIDEA-4 and weak subkeys concerning the second IDEA instance.

$Z_0 : 395d\mathbf{0000}0b1972f2_x,$	$Z_1 : 14687352eebfa3a3_x,$	$Z_2 : eda06123101bbcae_x,$
$Z_3 : 86a0\mathbf{0000}5b003dd8_x,$	$Z_4 : a386b84284567a12_x,$	$Z_5 : a469c03e47fbb15b_x,$
$Z_6 : 7324\mathbf{0000}8b1de93a_x,$	$Z_7 : d7560a9fbb483f76_x.$	

Table 6. 512-bit user key of WIDEA-4 and weak subkeys concerning the third IDEA instance.

$Z_0 : 59c63711\mathbf{000}05ec_x,$	$Z_1 : 7352e14d3f6f1e79_x,$	$Z_2 : 04602ecc119a73b9_x,$
$Z_3 : 63fde770\mathbf{0001}8fc5_x,$	$Z_4 : 6ed237962c1a1f3a_x,$	$Z_5 : ff9dc3e17cd9f303_x,$
$Z_6 : 210a0fbc\mathbf{000}57a5_x,$	$Z_7 : 15e4659e944c63a5_x.$	

Table 7. 512-bit user key of WIDEA-4 and weak subkeys concerning the fourth IDEA instance.

$Z_0 : 1d4c31917771\mathbf{0001}_x,$	$Z_1 : d94189cbd37575e4_x,$	$Z_2 : d591a3515b7bee44_x,$
$Z_3 : 5f83fa2d40b7\mathbf{0001}_x,$	$Z_4 : d283c7edd6b1a0b2_x,$	$Z_5 : dc7670a5e493c19a_x,$
$Z_6 : b8efbcfe3d5c\mathbf{0001}_x,$	$Z_7 : aae76a1b39f1fe8e_x.$	

Table 8. 1024-bit user key of WIDEA-8 and weak subkeys concerning the second IDEA instance.

$Z_0 : 542d\mathbf{0000}bbd83bc8ac9707b155f63bd3_x,$	$Z_1 : 735266ee520753f77df9fab137eff93f_x,$
$Z_2 : be979d90686e51da5932877b69d63846_x,$	$Z_3 : f1d6\mathbf{0000}9cacff2b71662da75594117b_x,$
$Z_4 : 2a36af25e7789f0958478b75c263393e_x,$	$Z_5 : 5bac73e09b4c0b45de379e4d0cc563e5_x,$
$Z_6 : 6221\mathbf{0001}cf0172f9dd181aef3ff10779_x,$	$Z_7 : df41e88f0dd657efcc4345a3020f738b_x.$

Table 9. 1024-bit user key of WIDEA-8 and weak subkeys concerning the third IDEA instance.

$Z_0 : ce0369cc\mathbf{0001}1fa0112b402378607d8c_x,$	$Z_1 : 52d228190db8b2fb36fe7cb6a4923ff3_x,$		
$Z_2 : 7f60ec624ce8c9fda66e8e6025db6591_x,$	$Z_3 : e9052c24\mathbf{0001}b497ea4bde45c769b4eb_x,$		
$Z_4 : 5a78636ecb0bbf3fa4ce55b9e0a8f7f7_x,$	$Z_5 : 1449c25a95e64dc878d7522b28a2c65b_x,$		
$Z_6 : 526b94b7\mathbf{0001}2ce2af34521b35ec9470_x,$	$Z_7 : f3ffb172b7787e30bec74c2816a52b56_x.$		

Table 10. 1024-bit user key of WIDEA-8 and weak subkeys concerning the fourth IDEA instance.

$Z_0 : 271684859e1a\mathbf{0000}545c90f7861619c8_x,$	$Z_1 : 77c21d2c72e99573add29cdd403172cb_x,$
$Z_2 : c583ce79a6b3be73c2c9ec870eb33d8c_x,$	$Z_3 : 509a5df54e27\mathbf{0001}de2a1ba34aa04294_x,$
$Z_4 : feb294de96090aac23518277da3a01ed_x,$	$Z_5 : 41d5bf9ab09075d409f7ea767db1ff8b_x,$
$Z_6 : 40e086b9c8a8\mathbf{0001}4d617ed4298a2d05_x,$	$Z_7 : a42694d8640b5ffffd1dbeb2fb899de21_x.$

Table 11. 1024-bit user key of WIDEA-8 and weak subkeys concerning the fifth IDEA instance.

$Z_0 : dc46b3c130f2c632\mathbf{0000}f934498aa261_x,$	$Z_1 : 23c47bd870045f66226b5403b7818766_x,$
$Z_2 : 4bc944e3f9d5d7a801a7a3b11d54c07a_x,$	$Z_3 : 977fe5a7a2f23dea\mathbf{0001}3a585c5c4b6c_x,$
$Z_4 : dbbc32be55fd2b512ec881770ce640c2_x,$	$Z_5 : f7c60b939bf735e3db7b6cf6c403eea7_x,$
$Z_6 : 34d8c582b201fe40\mathbf{0001}4ce16e5feb64_x,$	$Z_7 : c9d2b3dfb6577915984e7cb86638ba4b_x.$

Table 12. 1024-bit user key of WIDEA-8 and weak subkeys concerning the sixth IDEA instance.

$Z_0 : 48175bed72710162acf2\mathbf{0001}c4ce4e1b_x,$	$Z_1 : 8697b91966d475e00d7cac9450f11e78_x,$
$Z_2 : 7a448cb7f2f7144078185192ad45b2c3_x,$	$Z_3 : 10994b4b70f6a90d2bd7\mathbf{0001}8f07f7e0_x,$
$Z_4 : 06eb0f2048330f57917d60215a5ff33a_x,$	$Z_5 : ed2c50d87fdf9a4de109750039e950bc_x,$
$Z_6 : 0155b95592ad5a8181c4\mathbf{0001}2d9888ab_x,$	$Z_7 : cbfd21bde970ba6e83bb27fe3b39ab4c_x.$

Table 13. 1024-bit user key of WIDEA-8 and weak subkeys concerning the seventh IDEA instance.

$Z_0 : ef54735c2c732a7fb4817009\mathbf{0000}d4d8_x,$	$Z_1 : c9c697819a33561fbe9b3fc9ebbb425c_x,$
$Z_2 : 81a4a2f1a1222eb8892b8af1bab30416_x,$	$Z_3 : 1d5869ad16ac69728f909e63\mathbf{0000}023a_x,$
$Z_4 : 34e6b1d745b0302f6011507ca8a2a0e4_x,$	$Z_5 : aa2fd64fc7ca837cad105a832d70536c_x,$
$Z_6 : 1ad19efc2df74cc3692fe9f1\mathbf{0001}edd3_x,$	$Z_7 : ec25fd09bd08537d589fff69871be4833_x.$

Table 14. 1024-bit user key of WIDEA-8 and weak subkeys concerning the eighth IDEA instance.

$Z_0 : dc7853502880fdf51dc9e795e226\mathbf{0001}_x,$	$Z_1 : 8089ea6c425c1be032fcf0123c779012_x,$
$Z_2 : 8c888bcc4f27c6d35338a6013b6a1ca6_x,$	$Z_3 : 3e619a3dcc1878be7a02a4767961\mathbf{0001}_x,$
$Z_4 : b4e89acbe72198f7a752385ab3961373_x,$	$Z_5 : 8bfdaa848980f7071bbaff068eaa6fce_x,$
$Z_6 : c1e72b1cac7ac68fb604ad52ba5e\mathbf{0001}_x,$	$Z_7 : 062046a11ac16410bd69f29368c0bffe_x.$