Logical Reasoning to Detect Weaknesses About SHA-1 and MD4/5

Florian Legendre* Gilles Dequen† Michaël Krajecki‡

Abstract

In recent years, studies about the SATisfiability Problem (short for SAT) were more and more numerous because of its conceptual simplicity and ability to express a large set of various problems. Within a practical framework, works highlighting SAT implications in real world problems had grown significantly. In this way, a new field called logical cryptanalysis appears in the 2000s and consists in an algebraic cryptanalysis in a binary context thanks to SAT solving. This paper deals with this concept applied to cryptographic hash functions. We first present the logical cryptanalysis principle, and provide details about our encoding approach. In a second part, we put the stress on the contribution of SAT to analyze the generated problem thanks to the discover of logical inferences and so simplifications in order to reduce the computational complexity of the SAT solving. This is mainly realized thanks to the use as a preprocessor of learning and pruning techniques from the community. Third, thanks to a probabilistic reasoning applied on the formulas, we present a weakness based on the use of round constants to detect probabilistic relations as implications or equivalences between certain variables. Finally, we present a practical framework to exploit these weaknesses through the inversions of reduced-step versions of MD4, MD5, SHA-0 and SHA-1 and open some prospects.

Keywords : Logical cryptanalysis, MD5, SHA-1, Satisfiability

Introduction

The SATisfiability Problem is a well-known decision NP-Complete problem [1], central in mathematical logic and computing theory. Over the last years, under its conceptual simplicity and ability to express a large set of various problems, the focus on SAT solving has

*UFR Sciences, University of Reims Champagne-Ardennes, Moulin de la Housse, Reims, France. This work is supported by the Direction Generale de l'Armement (DGA) from France. E-mail: florian.legendre@univ-reims.fr

†MIS, University of Picardie Jules Verne, Amiens, France. E-mail: gilles.dequen@u-picardie.fr

‡UFR Sciences, University of Reims Champagne-Ardennes, Moulin de la Housse, Reims, France. E-mail: michael.krajecki@univ-reims.fr

significantly grown. To date, it remains a core problem in artificial intelligence, logic and computational complexity theory. For nearly 15 years, improvements dedicated on the one hand, to the original backtrack-search DLL procedure [2], and on the other hand to logical simplification techniques [3] have increasingly diversified and a new generation of SAT solvers succeeded in solving huge problems from industrial areas[1] [4]. Within a practical framework, SAT applications are as diverse as planning [5], model checking [6], VLSI design and also cryptography [7]. In addition, one challenge of the continuous technological development is to provide a robust security to computer systems. This generally relies on the use of cryptographic primitives, for which their guarantee are strongly correlated to their ability to preserve a secret (*i.e. plaintext* or *cipher key*). Within this framework, it is essential to study all the techniques that are interested in the security of a primitive with a view to finding weaknesses in it that will facilitate the retrieval of any secret information. Several types of cryptanalysis approaches exist (linear [8], differential [9], Meet-in-the-middle attack [10], side channel attack [11], etc...) and each one is efficient following the tackled cryptographic function. In this paper, we focused on a specific algebraic cryptanalysis called *logical cryptanalysis* which takes advantage of the remarkable progresses of the SAT community. This was first described in the early 2000s in [12] where the idea was to find out the cipher key of the Data Encryption Standard (DES) by encoding the original stream cipher as a SAT problem to instancing the propositional variables corresponding of input/output plaintexts. From this, more and more works emerged in the logical cryptanalysis field allowing to create difficult SAT formulas for benchmarking [13, 14, 15] or to analyze the resistance of cryptographic functions against the power of SAT solvers.

In this last point of view it can be dissociated the use of the SAT formalism to accelerate some specific computation in certain attacks to the ones which are totally devoted to a SAT-based attack. In the first approach, [16] present an interesting result about cryptographic hash functions. The authors presented a new application of logical cryptanalysis assuming that the runtime of a cryptanalytic attack should be improved by using a logic formalism to express complex operations. Practically, they modeled a whole differential path for the best known hash functions (MD[4/5] and SHA-[0/1]) into a boolean circuit and obtained conclusive results by using some of the best SAT engines. In this research field, it exists also some interesting works about symmetric and asymmetric primitives as for instance in [17, 18, 19, 20]. With regard to the works about measuring the security of cryptographic hash functions thanks to logical cryptanalysis, most of the studies are practical preimage attacks lead about MD4, MD5, SHA-1 and SHA-3 candidates [21]. Generally, the angle of attack which is privileged is a two-steps process following a boolean modeling and then a dedicated SAT solving. The modeling phase is to express, *in extenso* and independently from the solving phase, the algorithmic process associated to a cipher primitive, a hash function or more generally a dedicated attack, to a set of boolean equations (a SAT

[1] http://www.satcompetition.org

formula). It then results a SAT formula describing the whole process where the sequentiality disappeared and for which it exists at least one assignment of its variables in such a way as to make it evaluate to TRUE. To made this type of SAT formulas, some tools exist[2] as for instance *Grain of Salt* [3] or *cryptologver* [21]. An other method is to handcraft the SAT formulas thanks to program exclusively dedicated to the tackled primitive as we processed in this paper. Once a SAT formula is generated, the solving phase is to instantiate a hash and then finding a solution thanks to a SAT solver. Within a practical framework, reduced versions of the primitives are tackled to give a bound that measures the resistance to preimage attacks. At last, should be noted that within complexity theory context the co-NP side of the SAT problem which consists in proving the non-existence of any solution is practically absent unless to modeled corrupted cryptographic processus. Logical cryptanalysis is very hopeful however in its current state it still has some limits. Indeed, the used SAT solvers are made to answer to generic (and industrial) problems and so are not especially designed to treat cryptographic problems which yet contain very particular structures. This means, it paradoxically brings a new interesting factor to measure the security of a cryptographic primitive without consider the problem it is dealing with. Furthermore, although difficult to implement, these preimage attacks don't generally take into account of any other cryptanalytic cognition. In our knowledge, the only exception is in [22], where the authors tackled the second preimage of a 39 steps (about 48) reduced version of MD4 by adding some information from the Dobbertin's attack [23].

1 Contribution of this paper

In this paper, we are first interested in a logical cryptanalysis of hash functions by two modes. First is using SAT formalism to provide an original understanding of hash functions and the second is tackling the inverting problem lead to (second) preimages. In this way, drawing on the primitives describing these functions, we show how to encode basic functional properties such as modular additions, {*xor, and, or*} operations and some non-linear functions. Our encoding is handcrafted, *i.e.* in practice we implemented our own generator and each operation has its own encoding. Contrary to the use of generic and automatic tools, mainly based on *Tseitin* [24] transformations, this approach seems to be the most adapted because we don't have superfluous variables or clauses. This point appears to be intuitive, since the more the modeling is close to the real problem, the more it is easier to solve it. A review about this point is conducted in section 7.2.

Second, we use our formulas to enrich our encoding. In this sense, a first work was made by using the SAT formalism as a logical reasoning tool. In fact, before thinking resolution, the SAT representation of a problem can be used to compress and enrich the problem

[2] ECRYPT II Tools for Cryptography - http://www.ecrypt.eu.org/tools/

[3] http://www.msoos.org/grain-of-salt/)

thanks to inferences rules. In this paper we explain all of these rules and show how we preprocessed our formulas to reduce the size of the problem and learn new information like assigned variables, implications and equivalencies relations. In this way, our formulas were used to provide our perspective on how SAT can helps cryptanalysis. SAT literature provides various preprocessing techniques that are not implemented in SAT solvers due to their time consumption. However, it exists some static tools that are applied before solving an instance but they don't include all the simplifications techniques we presented here. Finally note that, even if a static preprocessing consumes a substantial runtime, it generally allows to then ameliorate the solving of an instance, particularly if the modeled problem is very structured.

An interesting point is that formulas resulting from the modeling phase can be easily solved as a solution assignment corresponds to a hashing process. Within this framework, a lot of assignments can be found in an effortless way. Nevertheless, searching an assignment for a particular solution is very difficult.

In addition to this classical logical reasoning, we are also lead some studies about probabilistic logical reasoning. In this way, we propose an experimental protocol to compute probabilities about each variables composing the formulas in order to obtain a general assigning of the variables in correct hashing processes. An analysis of these probabilities was then realized to discover specific relations between certain variables. To describe them, we introduced the terms of *quasi-implication*, *quasi-equivalences* and *quasi-assigned variable* and give a concrete sample of them. Both of factual and probabilistic relations are pertinent information to add in our formulas because this enriches the modeling. We believe this could be an important aspect to take into account to solve the formula in a more efficient way. In this paper, we explain non-randomness of certain variables and so *quasi-relations* by a weakness about the use of round constant.

Finally, to put our diverse contributions into practice we tackled the searching of preimages about reduced versions of MD4, MD5, SHA-0 and SHA-1 and analyzed our results to conclude.

Plan of this paper

The paper is organized as follows. We first recall in section 2 some basic aspects about cryptographic hash functions, and give a detailed description of MD5 and SHA-[0/1]. Based on these information, we then propose a modeling of these primitives into a SAT formalism and yield some specifications about the way to encode.

We present two methodologies to gather information about the structures that are in our formulas. A first approach consists in a logical reasoning presented in section 4 to allow the detection of some specific simplifications. A second approach based on probabilities is highlighted in section 5. In this part, the stress is put on weaknesses due to the use of round

constant. These two sections evoke how to use the SAT expertise to, on the one hand, reduce the size of the generated formulas and on the other hand, learn new structural information about the analyzed hash function. Afterwards, a practical experiment is given in section 7.2 where the detected weaknesses are used through a concrete preimage attack. Finally, the paper concludes with some topics for future research.

2 Preliminaries

2.1 About cryptographic hash functions

A cryptographic hash function can be defined as a deterministic algorithm that maps an any-length bit string (also named *message*) to a fixed-length bit string, usually named *digest* or *hash*. Among the uses of such a function it can be noticed the integrity check of files or communications or digital signature. It can also contribute to ensure authentication protocols with Message Authentication Codes (MACs) which are a mean that two users with a shared secret key can authenticate between each other.

Our studies are mainly focused on the popular MD[4/5] and SHA-[0/1][4] hash functions that are built following the *Merkle-Damgard* construction [25, 26]. This construction consists in an iteration of a compression function $\mathcal{F}$ applied on each fixed-size blocs m_i from the input message m and which take as parameter an internal state H_i. The last computed internal state is the result of the cryptographic hash function.

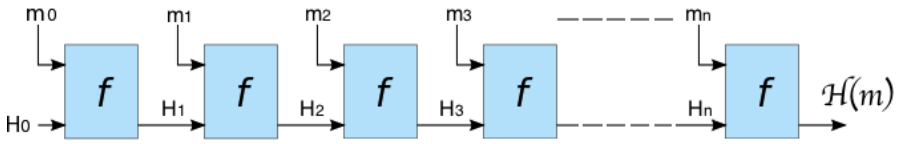

Figure 1: Merkle-Damgard construction

To ensure hash functions are secured, they required to be theoretically and computationally collision and (second) preimage resistant. A collision is when one can find two messages m and m' such as $\mathcal{H}(m) = \mathcal{H}(m')$. This attack is the easiest way to weaken (nay break) a hash function and supply many tremendous results [27, 28, 29, 30, 31, 32]. Unlike, with regard to preimage attack, the following summary may be given. Given a hash function $\mathcal{H}$ and a digest h, a preimage attack consists in finding a message m such as $\mathcal{H}(m) = h$. Even if it is the hardest way to tackle a cryptographic hash function, this is mostly the path chosen by SAT-based attack.

[4] In this paper, we use the notation MD[4/5] and SHA-[0/1] to respectively talked about both MD4 and MD5 and respectively of SHA-0 and SHA-1.

2.2 Notations

The studied hash functions in this paper are mainly MD5 and SHA-1. Because they are based on the Merkle-Damgard construction, they have some particularities in common that are defined in the same manner. These functions uses internal 32-bit words that are described with the following notations. Let be the process at the step i. We denote each word as:

- Q_i is the internal state obtained at the end of a step. It is the concatenation of four 32 bits-words in $\{A, B, C, D\}$ (resp. 5 in $\{A, B, C, D, E\}$) within the MD[4/5] (resp. SHA-[0/1]) process.
- f_i is a non-linear function. It can be named {F, G, H, I}
- S_i is a sum resulting of a four operands addition. This represents the main operation of a round.Within the propositional context, an addition of four operands could generate two levels of carry. fC_i is the first level. sC_i is the second level of carry.
- tC_i is the first (and unique) level carry resulting of a two operands addition
- Cst_i is a round constant
- M_k is the k^{th} 32-bit word from the input message, $k \in \{0, ...15\}$
- Finally, for any 32-bit word X, $X[j]$ denotes the j^{th} bit, with $j \in \{0, ...31\}$

2.3 Precisions about MD5

MD5 was designed in 1991 by Ron Rivest as an evolution of MD4, strengthening its security by adding some improvements. A description of the algorithm is given in the RFC1321[5]. This function is broken by collision attacks and the best of them has a complexity about $2^{20.96}$ [33]. Its preimage is also weakened by a theoretical attack with a complexity about $2^{123.4}$ [34]. The operating principle of this function is based on the well-known *Merkle-Damgard* model and consists in the repetition of 64 steps fairly distributed in four rounds, where one of them can be seen in the figure 2 and defined with three sub-steps as follows:

a) $Q_i \leftarrow Q_{i-4} + f(Q_{i-1}, Q_{i-2}, Q_{i-3}) + M_k + Cst_i$

b) $Q_i \leftarrow Q_i \lll s_n$

c) $Q_i \leftarrow Q_i + Q_{i-1}$

[5] http://www.ietf.org/rfc/rfc1321.txt

where :

- $i \in \{ 1, ..., 64 \}$, $k \in \{ 0, 1, ..., 15 \}$,
- $Q_{-3}, Q_{-2}, Q_{-1}, Q_0$ are the initial values (I.V.).
- $\lll R_n$ the circular shifting to the left(rotating) by n bits position, depending on i.
- The non-linear function f_i is defined by:

f_i = F(X,Y,Z) = (X $\wedge$ Y) $\vee$ ($\overline{X}$ $\wedge$ Z), $i \in$ [1, 16]
f_i = G(X,Y,Z) = F(Z,X,Y) $i \in$ [17, 32]
f_i = H(X,Y,Z) = X $\oplus$ Y $\oplus$ Z, $i \in$ [33, 48]
f_i = I(X,Y,Z) = Y $\oplus$ (X $\vee$ $\overline{Z}$), $i \in$ [49, 64]

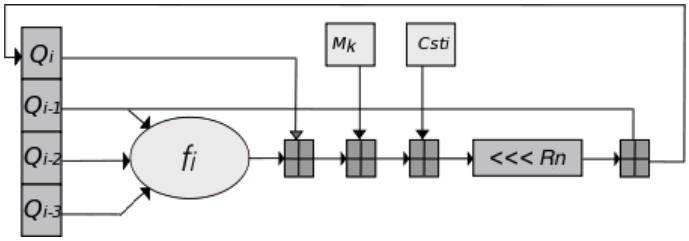

Figure 2: One MD5 step

2.4 Precisions about SHA-1

SHA-1 was designed in 1995 by the NSA as an improved version of SHA-0 in order to prevent some weaknesses. A description of the algorithm is given in the RFC4634[6]. The best attacks against a full SHA-1 are theoretical and presented in [35] where a complete collision attack with a complexity about 2^{61} and a near-collision attack with a complexity about $2^{57.5}$ are described. Its preimage is still considered not weakened. The operating principle is the same as the MD[4/5] family and consists in a hashing process where five states of 32-bit words are initialized and then modified at each of the 80 steps. One step of the SHA-1 process is detailed in the figure 3. Each of them can be defined with the following sub-steps:

a) $Q_i \leftarrow (Q_{i-1} \lll 5)$

b) $Q_i \leftarrow Q_i + f(Q_{i-2}, (Q_{i-3} \lll 30), Q_{i-4})$

c) $Q_i \leftarrow Q_{i-5} + W[i] + Cst_i$

[6] https://tools.ietf.org/html/rfc4634

where :

- $i \in \{ 0, 1, ..., 79 \}$, the current step.
- $Q_{-1}, Q_{-2}, Q_{-3}, Q_{-4}, Q_{-5}$ represent the initialization vector (I.V.).
- Cst_i is defined among four predefined constants.
- $\lll r$, the circular shifting to the left(rotating) by r bits position.
- The non-linear function f_i is defined by:

$$f_i = \text{F(X,Y,Z)} = (\text{X} \wedge \text{Y}) \vee (\overline{X} \wedge \text{Z}),\ i \in [0, 19]$$
$$f_i = \text{G(X,Y,Z)} = \text{X} \oplus \text{Y} \oplus \text{Z},\ i \in [20, 39]$$
$$f_i = \text{H(X,Y,Z)} = (\text{X} \wedge \text{Y}) \vee (\text{X} \wedge \text{Z}) \vee (\text{Y} \wedge \text{Z}),\ i \in [40, 59]$$
$$f_i = \text{I(X,Y,Z)} = \text{G(X,Y,Z)},\ i \in [60, 79]$$

- $W[i]$ is the i^{th} word of 32 bits, built from the input message as follows:

if $i < 16$
$W[i]$ is the i^{th} 32-bit word from the message.

if $16 \leq i \leq 79$
$W[i] \leftarrow (W[i-3] \oplus W[i-8] \oplus W[i-14] \oplus W[i-16]) \lll 1$

SHA-0 differs from SHA-1 by the shifting of $W[i]$. Note that contrary to SHA-1, SHA-0 is broken by a collision attack [36].

In the following of this paper, 1 round for a MD[4/5] (resp. SHA-[0/1]) function corresponds to 16 (resp. 20) steps.

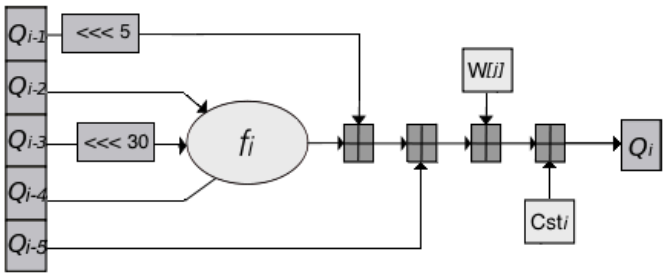

Figure 3: One SHA-1 step

2.5 Notations about SAT solving

Since our approach is based on the logical cryptanalysis principles, this work is closely related to SAT solving techniques. The last progresses about solving techniques have led SAT to be a great and competitive approach to tackle a wide range of industrial and academic problems as diverse as planning [5], model checking [6], VLSI design and also cryptography [7, 12] etc... Among this varied panel, logical cryptanalysis is a very recent use of SAT formalism.

More precisely, given a boolean expression $\mathcal{F}$, SAT is the decision problem of determining if $\mathcal{F}$ has at least one assignment of truth value (also named an *interpretation*) {TRUE, FALSE} to its variable so that it is TRUE. In this paper, $\mathcal{F}$ is considered as a CNF-formula (Conjunctive Normal Form) which can be defined as a set of clauses (interpreted as a conjunction) where a clause is a set (interpreted as a disjunction) of literals.
More precisely, let $\mathcal{V} = \{ v_1, ..., v_n \}$ be a set of n boolean variables. A signed boolean variable is named a *literal*. We denote, v_i and $\overline{v_i}$ the positive and negative literals referring to the variable v_i respectively. The literal v_i (resp. $\overline{v_i}$) is TRUE (also said *satisfied*) if the corresponding variable v_i is assigned to TRUE (resp. FALSE). Literals are commonly associated with logical AND and OR operators respectively denoted $\wedge$ and $\vee$. As mentioned above, a *clause* is a disjunction of literals, that is for instance $v_1 \vee \overline{v_2} \vee v_3 \vee v_4$. Hence, a clause is satisfied if at least one of its literals is satisfied. As a SAT formula $\mathcal{F}$ is considered under CNF, it is satisfied if all its clauses are satisfied. Finally, if its exists an assignment of $\mathcal{V}$ on {TRUE, FALSE} such as to make the formula $\mathcal{F}$ TRUE, $\mathcal{F}$ is said SAT and UNSAT otherwise.

In order to solve the SAT problem, two classes of techniques are commonly used by the community.

- *Incomplete* SAT solving methods are those that cannot guarantee an answer in a finite runtime. The relaxation of this guarantee leads these methods to practically behave as polynomial algorithms. Hence, depending on their success rate, they are able to answer more quickly than complete and enumerative techniques. In practice, such methods are dedicated to satisfiable formulas and are unfortunately not as good as complete methods to prove the unsatisfiability [37]. Moreover, they are intended to specific classes of problems such as randomly generated ones. Among the incomplete approaches to SAT solving, one of the most efficient is based on GSAT and WALKSAT algorithms which can be briefly describe as noisy and greedy searches into the search-space. The reader should refer to [38, 39, 40] for more details.

- *Complete* approaches guarantee an answer in a finite but exponential runtime. These methods are mainly based on the DPLL [2] algorithm which consists in a systematic enumeration of truth assignments generally thanks to a binary search-tree. A new generation of SAT solvers based on *Conflict Driven Clause Learning* CDCL [41, 42]

appears in the mid-90 and are particularly very effective in practical applications. The main improvement is based on a conflict analysis allowing to add a new clause to provoke backjumpings instead of classical backtrackings. We choose to use this type of solving approach within the context of this paper.

The figure 4 shows a comparison between these methods of resolution.

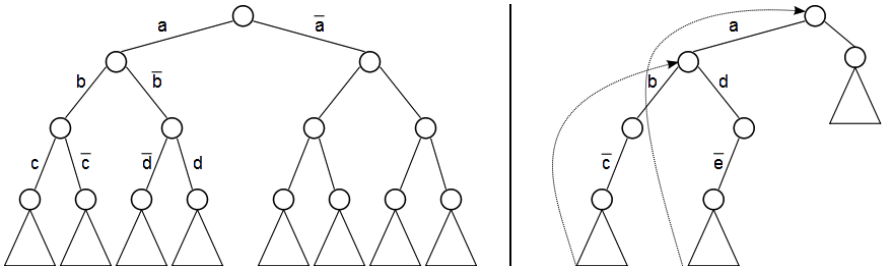

Figure 4: Comparison between a resolution thanks to a DPLL algorithm (on the left) and a CDCL procedure (on the right). In the first case, the DPLL procedure is a recursive enumerative feature with backtracking while the CDCL binary-search tree allows backjumping thanks to its clause learning.

The use of SAT formalism to express a cryptographic problem is a choice presenting some interesting advantages. For instance, a noteworthy point of logical cryptanalysis is the modeling leads to lost the notion of inherent sequentiality of the algorithm during the solving process. This phenomenon is due to the fact that the problem is rewritten under a set of clauses, which doesn't have any order. We propose to illustrate this crucial point with an instance of a 4-bits addition with holes. We denote by x_i the variable $x \in \{c, a, b, s\}$ at the index i.

Index	*3*	*2*	*1*	*0*	
car c	1	-	-	0	
op a	-	1	-	-	
op b	-	-	1	-	+
sum s	1	1	-	1	=

Figure 5: Holed addition of two binary operands

In a decimal framework, by looking at the figure 5 it is quite easy to see $a \geq 100_{(2)}$, $b \geq 10_{(2)}$, either a or b is odd and $s \in \{((1)1101)_{(2)}, ((1)1111)_{(2)}\}$. However, it is no so easy to export something from the row of the carries because c depends of a, b and c. Working in a binary field leads to have a very detailed notion of carries while it's not the case from a larger scale. Moreover, as the reasoning is logical it is possible to discover the value of

some other bits. Firstly, from c_0 and s_0, one can deduce $c_1 = 0$ and from c_3, a_2 and s_3, one can infer $c_2 = b_2 = 1$. This is described in the figure 6.

Accordingly, only four solutions stay possible:

$$(a, b) \in \{(0110, 0111)_{(2)}, (1110, 1111)_{(2)}, (0111, 0110)_{(2)}, (1111, 1110)_{(2)}\}$$

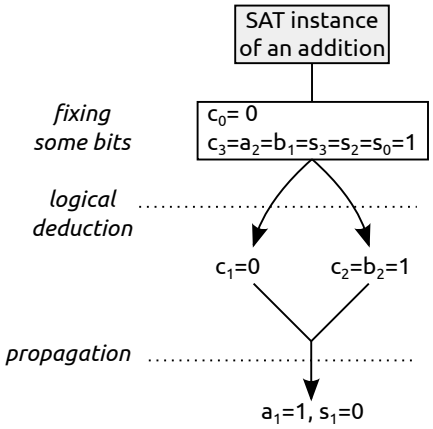

Figure 6: Deductions from logical inferences in the holed addition in figure 5

3 How to model a cryptographic function as a SAT problem

Logical cryptanalysis usually refers to a two-steps process where the first one consists in modeling a cryptographic function under a SAT formalism, and so the associated algorithmic process in boolean equations. This modeling can be done by two different ways.

A first way of seeing things, is to dedicate a part of the modeling to an automatic tool. This method is for instance used in [21] to encode SHA-3. In a first time, the authors translated the SHA-3 algorithm description into an SDL code and in a second time they used their SDL code in their own automatic tool called *cryptologver* to generate a CNF. *cryptologver* is a program relying on *Tseitin* [24] transformations to create CNF formulas from an SDL code in a easy way an so provide some non-negligible advantages. Indeed, the tool is generic and can be reused to any other cryptographic function. Consequently, both these advantages allow to have a simple method to analyze the robustness of a cryptosystem within a logical cryptanalysis framework. However, the price to pay for that is the modeling can be not optimized, contain redundant information and be unfortunately described in a non efficient version. For instance, the resulting CNF formula can be constituted of superfluous variables and clauses that are too deeply integrated to the model to then been spotted and removed by a preprocessing. In our point of view, this could result in a problem that is more difficult to

solve than it really is. To relieve this, an other method involves modeling a cryptographic function by cutting it following its atomic operations, model these different pieces in a logical formalism and then put them together to reconstruct the encryption algorithm. In other words, this means to separately describe the different operations of such an algorithm as a SAT problem and then make all pieces of the puzzle fit together. In our works, we chose this way to represent the cryptographic hash functions we studied. Even if few works exists on this subject, we suppose that a good modeling can be crucial to decrease the runtime of the solving phase for structural SAT instances.

Regarding the cipher primitives of SHA-[0/1] and MD[4/5], hashing processes are an assemblage of several steps, each one composed of basic operations as for instance modular additions, circular shiftings (or rotating) to the left and some booleans operations. In the next parts, we distinctly give an encoding of each of the components that occur in both of these hash functions. A detailed description of the method is given and the choices we have to take are justified.

3.1 The modular addition of n operands

The modular addition of n operands is invoked in many cryptosystems and particularly in SHA-1 and MD5. It consists in a classical addition of n fixed-length of m-bits operands. This addition is defined in a Galois field GF(2^m). To describe it, we first focus on the easiest case, a modular addition of two 1-bit operands.

3.1.1 The modular addition of 2 operands

To implement this basic operation, we choose to directly express into SAT clauses the logical rules associated to the classical arithmetical addition. In this sense, and considering a simple adder circuit, this can be seen as two operations of implications: (operands) $\Rightarrow$ (sum) and (operands) $\Rightarrow$ (carries). A modeling of this simple adder is presented on figure 7, where s_i corresponds of the sum of a_i and b_i. The black arrow represents the likely generation of a carry noted c_{i+1}. This carry must be considered as an other operand at the rank $i+1$. Consequently, to define a generic representation for a modular addition of two operands we necessarily need to consider a modular addition of three operands: a_i, b_i and c_i, with c_i coming from the rank $i-1$ (except at the rank $i = 0$).

Based on this model, the associated boolean truth table is set up. It represents all the possible outputs which are generated considering each possible combinations in input. In other words, it describes the inference rules that define the classical reasoning of an addition of two operands. In white cases of the figure 8, the operands are input variables and in light gray are the output variables.

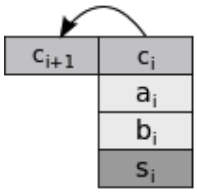

Figure 7: Model of a modular addition of two 1-bit operands

c_i	a_i	b_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Figure 8: Truth table of an addition of two 1-bit operands

From this truth table, we conclude in a generation of SAT equations, where each line corresponds to two clauses (one clause to each output variable). For instance, let c_i, a_i, b_i, c_{i+1}, s_i be five boolean variables, the second line of the table in figure 8 can be read as $c_i = 0$, $a_i = 0$, $b_i = 1$ implies $c_{i+1} = 0$ and $s_i = 1$. This can be formulated as follows:

$$(\overline{c_i} \wedge \overline{a_i} \wedge b_i \Rightarrow \overline{c_{i+1}}) \wedge (\overline{c_i} \wedge \overline{a_i} \wedge b_i \Rightarrow s_i)$$

And then, under a CNF:

$$(c_i \vee a_i \vee \overline{b_i} \vee \overline{c_{i+1}}) \wedge (c_i \vee a_i \vee \overline{b_i} \vee \ s)$$

3.1.2 Modeling the modular addition of n operands

To implement an n operands addition, the method is the same as the one considered for a two operands addition. The only difference resides in the fact that it outputs more levels of carries. The consequence is, as for a two operands addition, carries in output must be considered as inputs in the next corresponding rows. Therefore, a four operands addition (as the ones in MD5) becomes a six operands addition (See figure 9).

By generalizing this principle, an n operands addition, with $2^k - k + 1 \leq n < 2^{k+1} - k$, generates k vectors of carries.

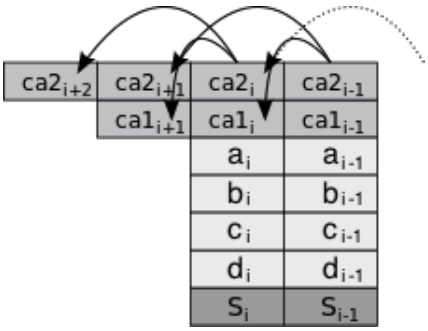

Figure 9: Model of an addition of four operands

3.2 The non-linear functions

Several non-linear functions are anchored at each steps of a cryptographic hash function, taking several variables as parameter to give one output. They are very often described in a boolean format and are another mean to give rise to chaos and strengthen to the cipher. By definition, these functions are encoded into a logical formalism and so it is sufficient to just translate them under a CNF format.

3.3 The circular shifting (rotating)

A circular shifting doesn't require any concrete description to be implemented as a SAT problem, *i.e.* it doesn't need any clauses or variables to be expressed. Generally, a circular shifting is used after an other operation. For instance, for MD5 and SHA-1 it is done after a four (respectively five) operands addition. Our idea is to model this circular shifting within this next operation thanks to a good encoding of the variables.

Let be R the result of the previous operation and $\lll s$ a circular shifting about s bits position. Within an algorithmic framework, first R is computed, then R is shifted to obtain $R_{\lll s}$ and finally $R_{\lll s}$ is used in an other operation. Thanks to a SAT formalism this can be represented in only one operation by concatenating the shifting of R and the next operation were it is used as an operand. An illustration of this example is on the figure 10 where a word R is shifted by 2-bits position to the left and then a two operands addition between $R_{\lll s}$and a 32-bits word A is computed. In practice this is modeled in only one bloc.

Finally, note that a circular shifting s applied on a m-bit word to the right is equivalent to a circular shifting to the left of $m - s$ bit positions.

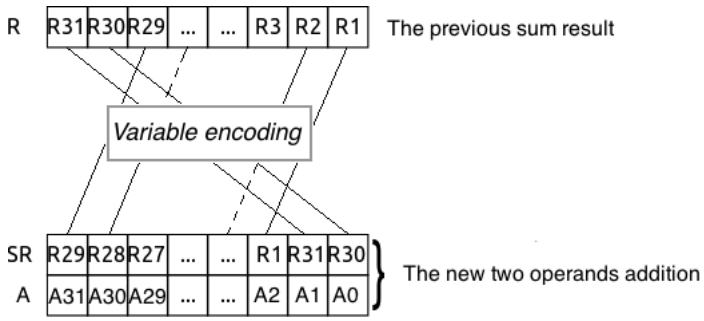

Figure 10: Model of a directly shifted addition

3.4 Application to MD[4/5] and SHA-[0/1]

To finally model in practice a cryptographic hash function all of these bricks are cemented during the generating process of clauses thanks to a correct variable encoding. Following the previously described method, we implemented a program that is able to provide CNF formulas for the whole MD4, MD5, SHA-0 and SHA-1 primitives by considering a fixed-length input of 512 bits (one block). This generator is configurable to assign a digest or an input message (or just some bits) and to provide reduced-step formulas. It's also possible to hash a message and obtain its digest. In the table 1 some specifications of the complete representations of these functions are given. A comparison between our CNF and the ones of the literature could be interesting but generally such information are missing.

Function	Clauses	Variables	Literals
MD4	144 885	6 690	824 378
MD5	224 643	12 749	1 236 634
SHA-0	491 791	12 779	3 283 930
SHA-1	491 791	12 779	3 283 930

Table 1: Statistics about the number of clauses, variables and literals required to represent under a SAT problem the cryptographic hash functions MD4, MD5, SHA-0 and SHA-1 using our methodology.

It is interesting to remark that both SHA-0 and SHA-1 have exactly the same statistics. This is explain by the fact that the only change between SHA-0 and SHA-1 is an add of a circular shift to the left during the diversification of the input message. Or, within a SAT context, a circular does not need any variable or clauses to be expressed.

4 Logical reasoning

The SAT community is able to evaluate the complexity of a random SAT formula by regarding the number of variable (n) and the number of clauses (m). However, giving a precise notion of complexity for a non-random SAT formula is impossible. In a general case, the complexity of this type of formulas is just bounded by regarding three parameters: n, m and the number of literals (l). Within this context, an upper bound is evaluated to $2^{\alpha\, n}$, with $\alpha < 1$, depending on n and m [38]. Consequently, to hope decrease this complexity, a good approach is to reduce the values of these two parameters. In this part we show how an appropriate logical reasoning applied on our CNF allows to reduce the complexity. In addition, we show some interesting relations of equivalences between certain variables of the hashing process that could be used in an other cryptanalytic approach.

An important aspect of a our SAT modeling is that the representation is made at the scale of the bit. The different operations are thus considered into their atomic existence by taking into account all the possible cases. In this way, the modeling phase was done exhaustively and it seems intuitive to smooth the generated SAT formulas thanks to a preprocessing. To statically reduce the statistics of a SAT formula, the SAT community provides some preprocessing tool as for instance SATELITE [43] or HYPBINRES [44]. The used techniques in such tools are very interesting to compact a formula to its littlest expression by implementing efficient logical techniques. However, in our point of view a preprocessing can also help to add new clauses in order to enrich the solving phase by, for instance, adding pertinent binaries clauses. In this way, a preprocessing could be able to reduce and enrich a SAT formula by new information. It's why we developed our own preprocessing in order to decrease the practical complexity of the formula by combining classical simplifications and new techniques to create logical bridges to help a subsequent SAT solving. In this part we present all of these logical simplifications and give as result a set of equivalences detected from an application on our cryptographic SAT formulas.

4.1 Classical simplifications

SAT can be seen as a tool allowing to express any problem with boolean equations. The solving of such a problem is achieved with a dedicated SAT engine (also named *solver*) that deals with reasoning techniques from Artificial Intelligence (A.I). Within a solving framework, problems are treated by adding pertinent clauses and removing redundant information (clauses, variables, literals) thanks to fast logical simplifications.

Consider the formula $\mathcal{F}$ having the four following clauses, and look at three interesting logical simplifications:

$$\mathrm{c}_1 = (b \vee \overline{c}) \qquad \mathrm{c}_2 = (b \vee \overline{c} \vee f) \qquad \mathrm{c}_3 = (e \vee f \vee \overline{g}) \qquad \mathrm{c}_4 = (e \vee f \vee g)$$

- Observe that c_2 is equal to $(c_1 \vee f)$. In this case, c_2 contains as much information as c_1 and if c_1 is satisfied, necessarily c_2 is satisfied too. This well-known process is called a *subsumption*, and since c_1 *subsume* c_2, c_2 is withdrawn.

- Now focus on c_3 and c_4 and note these clauses differ only in the signedness of the variable g. This scheme is known as *resolution*. From this, a new clause named *resolvent* is generated and is composed of all the variables of the two previous clauses except the one that differs in signedness. In our example, the clause $c_5 = (f \vee g)$ is added and in that case, c_5 *subsumes* both c_3 and c_4 and could be helpful within the solving.

The structure of the modeled problem makes that these two simplifications occur a very large number of times to reduce significantly the statistics of the CNF formula. For instance, 192 clauses and 1 344 literals are inventoried for an exhaustive modeling of a 1-bit four operands addition. Here, the model counts only 128 smaller clauses and 832 literals. The gain is about 36% on the number of clauses and 32% on the number of literals, and is approximately the same about an extension to a four 32-bits operands addition.

4.2 Static Look-Ahead and more

4.2.1 Principles

We also imported a classic local treatment in SAT solving named *Look-Ahead* [45] which consists in foreseeing the effects of choosing a branching variable to evaluate one of its values. In other words this corresponds to answer the question: what happened is a variable v, and only v, is set to true (or false). From this evaluation, it can infer an assignment or some information among equivalences, fixed literals and new binary clauses. Hereafter, some details.

- *Fixed literals*:

  i) if $a \Rightarrow false$ then $\overline{a}$ must be set to TRUE

- *New binary clauses:*

  ii) if $a \Rightarrow b$ then the clause $(\overline{a} \vee b)$ can be added to $\mathcal{F}$. This will be same if $(a \wedge b)) \Rightarrow false$ occurs.

  iii) if $a \Rightarrow (x_1 \wedge x_2 \wedge \ldots \wedge x_n)$ and $\overline{a} \Rightarrow (y_1 \wedge y_2 \wedge \ldots \wedge y_m)$ then clauses $(x_i \vee y_j), \forall (1 \leq i \leq n)$ *and* $(1 \leq j \leq m)$ can be added to $\mathcal{F}$.

- *Subsuming Look-Ahead [46]*

By enhancing the principle of look-ahead, one check multiple implications in order to produce subsuming clause.
Let be C a clause of the form : $x_1 \vee x_2 \vee \ldots \vee x_i \vee \ldots \vee x_k$.

iv) if $\overline{x_1} \wedge \overline{x_2} \wedge \ldots \wedge \overline{x_i} \Rightarrow false$ then C should be replace by the clause (*is subsumed by*) $x_1 \vee x_2 \vee \ldots \vee x_i$ in $\mathcal{F}$.

v) if $(x_1 \wedge \overline{x_2} \wedge \ldots \wedge \overline{x_i}) \Rightarrow false$ then C is subsumed by $x_2 \vee \ldots \vee x_i$

4.2.2 Detection of equivalences

The existence of a logical equivalence, from a point of view of a valid hashing process, means that at least two digits (it could be more) are linked by their respective value in every model. Practically and informally, this can be seen as a digit that has *always* the same (or opposite) value as another one. If such a case occurs, both digits represent the same information and only one of them should be considered into the process. Such a relation is denoted with the operator: $\Leftrightarrow$. As an example, consider the CNF formula $\mathcal{F}$ having the following clauses, the *detection of equivalencies* can be computed as :

$c_1 = (a \vee \overline{b})$ $\quad c_2 = (\overline{a} \vee \overline{f})$ $\quad c_3 = (b \vee \overline{c})$ $\quad c_4 = (c \vee \overline{d})$
$c_5 = (f \vee \overline{g})$ $\quad c_6 = (g \vee \overline{h})$ $\quad c_7 = (a \vee d \vee \overline{e})$ $\quad c_8 = (\overline{a} \vee h \vee e)$

- If a is set to FALSE then it can directly be deduced that b, c, d and e must be set to FALSE, unless to falsify $\mathcal{F}$, thanks to the clauses c_1, c_3, c_4 and c_7 respectively. Hence, a equals to FALSE implies e equals to FALSE. We denote this implication $\overline{a} \Rightarrow \overline{e}$. The corresponding CNF expression is $\overline{e} \vee a$. As a remark, this clause also represents the implication $e \Rightarrow a$.
- In the same way, if a is set to TRUE then f, g, h are set to FALSE and e to TRUE.
- Consequently, since $e \Rightarrow a$ and $a \Rightarrow e$, this means that whatever the solution, a and e have the same value. This is denoted $a \Leftrightarrow e$. Consequently, one can substitute every e in the formula by a (and vice versa). From this, may result a cascade of new simplifications. For instance, proceeding that substitution in $\mathcal{F}$ leads c_7 and c_8 to become obsolete, $a \vee \overline{a}$ being tautological and therefore useless.

4.3 Application to hash functions: the case of MD5

Several equivalences result from applying dedicated treatments on our SAT formulas. Some of them are trivial and others are not so. In the following, some examples are mentioned.

- *A trivial case:*
  $F_1[29] \Leftrightarrow Q_1[29]$. This equivalence is quite easy to detect because $F_1 = (Q_1 \wedge Q_0) \vee$

$(\overline{Q_1} \wedge Q_{-1})$ and Q_0 and Q_{-1} are I.V. and hence are constant. This means that if $Q_0[i]$ differs from $Q_{-1}[i]$ ($i \in \{0 \dots 31\}$), then F_1 depends exclusively on Q_1. There is a relation of equivalence which appears once on four on average between F_1 and Q_1.

- *Non trivial case:*
  This is the most interesting case. It seldom occurs within a general framework, but it gives pertinent information to cryptanalysts. We call *special case* a non-trivial case which occurs in a specific formula. For instance, if a treatment is applied on a CNF formula describing a preimage attack on MD5, then the equivalences that are exhibited are not related to the entire MD5 process but only to a specific instance. Thanks to that, by considering a preimage attack on the 29 first steps of MD5 where the digest is set to 0, it can be deduced: $M_8[2] \Leftrightarrow \overline{Q_{24}[2]}$, where M_8 is the 9^{th} bloc of the input message M.

- *Direct Implication:*
  If two implications of the form $a \Rightarrow b$ and $\overline{a} \Rightarrow b$ occur then for all value of a, b equals to TRUE. Consequently, b must not be set to FALSE unless to falsify $\mathcal{F}$. As an illustration, $sC_{39}[0]$ must be set to FALSE.

In the table 2, a concrete set of equivalences detected from our MD5 and SHA-1 SAT formulas is given. Note that these equivalences can be used in any other cryptanalytic approach.

All of these logical deductions simplify and enrich our SAT formulas. In the table 3, new statistics about the number of clauses and variables are given. It should be noted that these data also take into account binary clauses. These specific clauses are very interesting because treating them during the SAT solving is polynomial and make some logical bridges to accelerate the runtime.

Because we thought this learning is very helpful to help SAT solving, we also focused on a probabilistic approach to even more grind our cryptographic formulas in order to get new pertinent data.

5 Specific probabilities

In this section, an experimental framework is defined to allow a probabilistic study of a hashing process on its variables. Through this work, we define some *quasi-relations* between variables, *i.e.* a relation coming from a probabilistic reasoning, and show how to detect them. Moreover, we explain the mainspring of these particular relations by the use of round constant.

Equivalences from MD5
$\overline{tC_0[0]} \Leftrightarrow Q_1[0], f_1[0] \Leftrightarrow Q_1[0], \overline{S_0[25]} \Leftrightarrow Q_1[0], \overline{f_1[1]} \Leftrightarrow Q_1[1]$
$\overline{S_0[0]} \Leftrightarrow fC_0[0], \overline{f_1[2]} \Leftrightarrow Q_1[2], \overline{f_1[4]} \Leftrightarrow Q_1[4], \overline{f_1[5]} \Leftrightarrow Q_1[5], \overline{f_1[6]} \Leftrightarrow Q_1[6]$
$M_0[0] \Leftrightarrow fC_0[0], f_1[8] \Leftrightarrow Q_1[8], f_1[9] \Leftrightarrow Q_1[9], \overline{f_1[10]} \Leftrightarrow Q_1[10], \overline{f_1[12]} \Leftrightarrow Q_1[12]$
$f_1[13] \Leftrightarrow Q_1[13], \overline{f_1[14]} \Leftrightarrow Q_1[14], f_1[16] \Leftrightarrow Q_1[16], \overline{f_1[17]} \Leftrightarrow Q_1[17]$
$f_1[18] \Leftrightarrow Q_1[18], \overline{f_1[20]} \Leftrightarrow Q_1[20], \overline{f_1[21]} \Leftrightarrow Q_1[21], f_1[22] \Leftrightarrow Q_1[22]$
$f_1[24] \Leftrightarrow Q_1[24], f_1[25] \Leftrightarrow Q_1[25], f_1[26] \Leftrightarrow Q_1[26], \overline{f_1[28]} \Leftrightarrow Q_1[28]$
$f_1[29] \Leftrightarrow Q_1[29], f_1[30] \Leftrightarrow Q_1[30], \overline{Hb[0]} \Leftrightarrow cHb[0], \overline{Ha[0]} \Leftrightarrow cHa[0]$
$\overline{cHd[1]} \Leftrightarrow Hd[1], \overline{Q_{62}[1]} \Leftrightarrow Hd[1], \overline{cHc[1]} \Leftrightarrow Hc[1], \overline{Q_{63}[1]} \Leftrightarrow Hc[1]$
$\overline{Q_{64}[0]} \Leftrightarrow Hb[0], \overline{Q_{61}[0]} \Leftrightarrow Ha[0], Hd[0] \Leftrightarrow Q_{62}[0], Hc[0] \Leftrightarrow Q_{63}[0]$

Equivalences from SHA-1
$\overline{sC_0[28]} \Leftrightarrow fC_0[28], \overline{sC_0[27]} \Leftrightarrow fC_0[27], \overline{sC_0[25]} \Leftrightarrow fC_0[25], \overline{sC_0[24]} \Leftrightarrow sC_0[17]$
$\overline{sC_0[21]} \Leftrightarrow fC_0[21], \overline{sC_0[20]} \Leftrightarrow fC_0[20], \overline{sC_0[19]} \Leftrightarrow fC_0[19]$
$\overline{sC_0[15]} \Leftrightarrow fC_0[15], \overline{sC_0[12]} \Leftrightarrow fC_0[12]$

Table 2: Some of Equivalences detected from the MD5 and the SHA-1 processes. The used notation is the one presented in 2.2. Moreover Hx and cHx are variables about the hash.

5.1 From a SAT formula to get statistics

The SAT formula $\mathcal{F}$ is in fact an other way to express a hashing process. Thus, the assignment of variables corresponding to the input message leads the SAT engine to fix all the unassigned variables of $\mathcal{F}$ thanks to a linear and deterministic process named *unit propagation*. From this, it results a complete assignment $\mathcal{A}$ of the variables that corresponds to a solution of $\mathcal{F}$. Actually, $\mathcal{A}$ gives useful information on how a variable is set but also how two variables are set in pairs (and more). This last remark is interesting because it allows to appreciate the probabilistic behavior of a variable with respect to another. For instance, let v and w be two boolean variables. From $\mathcal{A}$, looking at v and w at the same time lets to know which of the following subsets of $\mathcal{S}$ appears:

$$\{v = false \wedge w = false\} \qquad \{v = false \wedge w = true\}$$
$$\{v = true \wedge w = false\} \qquad \{v = true \wedge w = true\}$$

respectively denoted $(\overline{v} \wedge \overline{w})$, $(\overline{v} \wedge w)$, $(v \wedge \overline{w})$, $(v \wedge w)$.

MD4				
	Clauses	**Variables**	**Literals**	**2-cl**
Before pp	144 885	6 690	824 378	2 317
After pp	108 081	6 673	531 100	3 154
MD5				
	Clauses	**Variables**	**Literals**	**2-cl**
Before pp	224 653	12 749	1 236 634	381
After pp	171 235	12 721	814 096	1 305
SHA-1				
	Clauses	**Variables**	**Literals**	**2-cl**
Before pp	491 791	12 779	3 283 930	259
After pp	375 195	12 771	2 210 708	908

Table 3: Comparison between statistics about the number of clauses, variables and literals to represent the MD4, MD5 and SHA-1 hash functions, before and after having applied an effective and dedicated preprocessing. The column called 2-cl is the number of clauses with only 2 literals.

From these statements, our idea was to establish a protocol to compute statistics from a SAT formula $\mathcal{F}$. This protocol is defined following five steps:

i) Create a random input message

ii) Assign this message and infers from $\mathcal{F}$. It generates $\mathcal{A}$.

iii) From $\mathcal{A}$, for each pairs of variables, memorize the subset which appears in $\mathcal{S}$

iv) goto i) (This loop should be iterated n times)

v) Group and overlay all the subsets and divide by n.

From this, we obtain a statistical database of the probability to be 1 for each couple of variables (v,w), denoted $\mathrm{p}(v \wedge w)$.

5.2 Preliminary remarks

The probability of a variable v to be 1 in a general framework is determined by:

$$\mathrm{p}(v) = \mathrm{p}(v \wedge v)$$

The conditional probability of a variable v given w is determined by:

$$\mathrm{p}(v|w) = \frac{p(v \wedge w)}{p(w)}$$

5.3 Generic probabilities about MD5

Applied on our SAT formulas the previous defined protocol allows to obtain a statistical database, where the probabilistic average of each variable to be assigned to 0 or 1 can be observed. However, this observation is difficult to conceive because of the high number of variables in the SAT formulas. In this part, we present a practical case on the cryptographic MD5 hash function to represent, and then use, this statistical database in a cryptanalytic framework. For this, an image was dressed where each pixel is a comparison between the computed probabilities in practice and the expected ones in theory.

Concretely, we computed for each variable the theoretical probabilities we should have in a general process. Then, we compared these data with our practical probabilities. For that, a PGM[7] image representing this comparison was drawn, vertically sort by step and horizontally sort by type of words (see figure 11). The used notation is the one defined in 2.2. On this image, if a pixel is black, the probability derived from the SAT formula differs by the theoretical probability by tending to 0, and if a pixel is white, the probability differs by tending to 1. Consequently, the more the pixel is white or black, the more the differential between the practical and the theory is high.

5.3.1 First analysis

On the face of it, it appears that the gray is not uniform in the columns representing the internal states (Q), the non-linear functions (f), the carries of the two operands addition (tC) and the four operands sum (S). An interpretation of this is the bits composing these words have a practical probability which is around $\frac{1}{2}$, as expected in theory. This means, the given assignments to the variables have a totally random behavior. However, this is not the case for the columns representing the carries of the four operands addition. Hereafter, some details and explanations.

5.3.2 About carries

The theoretical probabilities to be 1 for the first carry[8] turns around 0.58. Focus on step 17, we get for instance p($fC_{17}[5]$) = 0.67 and p($fC_{17}[23]$) = 0.49. The gap with the expected ones in theory is approximately 15%. This is important but stays however minor in comparison with the gaps we got by looking at the second carries.
Figure 12 zooms in the second carries vector from step 17. By taking a close look, probabilities are not uniforms (because there are several gray tones) and some go far away of their theoretical probabilities by tending to 0 (as for instance the 7^{th} bit) or to 1 (for instance

[7] Portable GrayMap file format

[8] Except the five first least significant bit where the probabilities are slightly higher. The details of theoretical values are provided in the appendices.

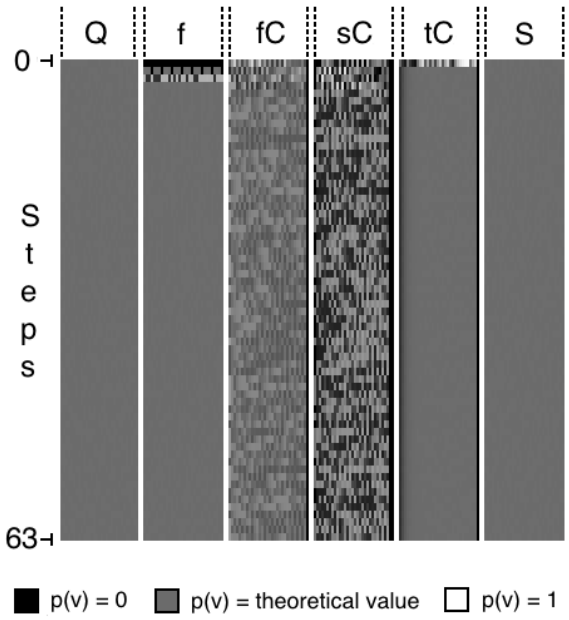

Figure 11: Figure representing a differential vector of probabilities on variables from the MD5 process, such as each pixel is about P(v), from step 0 to 63, sorted by type of 32-bits word on big endian.

Figure 12: Second carries vector from step 17, on big endian

the 22nd bit).

To explain these phenomenon, our assumption is the use of round constants at each step. The idea is that fixing a round constant consists in assigning an operand in the general structure of the addition and a consequence of this is the creation of a new structure totally configured by the round constant. It seems like an unstructured addition. To confirm this point, we computed theoretical probabilities of the variables which are involved in this addition by considering one operand fixed in the same way that in MD5. For instance, by looking at the step 17, an operand was concretely fixed to the value from the MD5's *RFC*[9], *i.e.*`0xc040b340`. We then computed again probabilities for the carries and observed how they behaved compared to those before. To analyze that three curves representing the probabilities in theory (in solid line), in practice (in little points) and in theory with a fixed constant

[9]RFC 1321 - The MD5 Message-Digest Algorithm: http://www.faqs.org/rfcs/rfc1321.html

(in dashes) were drawned on the figure 13. In abscissa, the place of the bit in the 32-bits word and in ordinate, probabilities go from 0 to 1. Looking at these curves, it can be noted that the probabilities in practice are very closed to the ones observed when we fixed a round constant because their respective curves are quasi superposed.

Consequently, this experimentation confirms our assumption consisting to say that using round constants weakened some words of the hashing process. This implies that internal words of an MD5 process are not entirely random and can be exploited within a practical attack. As an illustration, the bit 7, step 17 has a probability of 0.10 instead of 0.31 in theory and the bit 22 has a probability of 0.41. Finally, we think this phenomenon may help the discover of differential paths in collision attacks.

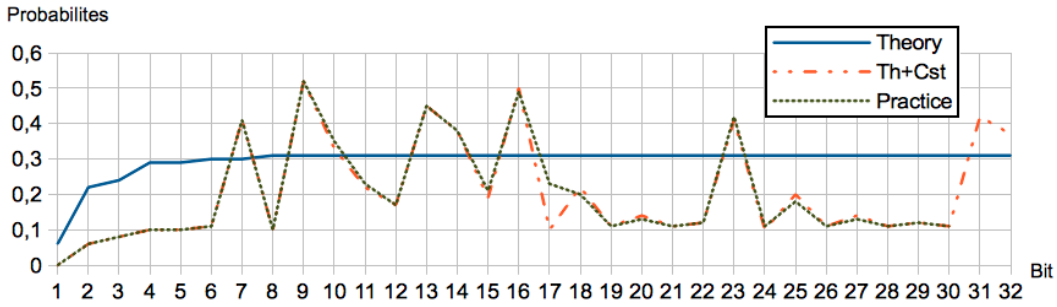

Figure 13: Comparison between theoretical probabilities with (*Th+Cst*) and without (*Theory*) a fixed round constant, and computed probabilities in our benchmark (*Practice*). Here we can observe that the curves *Th+Cst* and *Practice* are grouped. This means, they have the same probabilistic behavior, which differs from the one we could obtained in theory without any round constant.

5.3.3 Conditional probabilities

Conditional probabilities have also been represented on a PGM image in regarding the probability of a variable v given the probability of $v1$, *i.e.* $p(v|v1)$ (figure 14). Looking at this, probabilities for the carries are a somewhat strongly pronounced. This could be intuitive because carries depends on the previous rank but it shows that given a variable, probabilities of their local neighbors are even more influenced. These information could be pertinent in the case where an attacker is aware of some bits from the internal process. Thus, he might deduces how behaves a correlated variable with these known bits. Moreover, some other interesting probabilities can be computed to help cryptanalysts, notably in the searching of good differential paths by computing $p(v_i|v_{i-1})$, where i is a step.

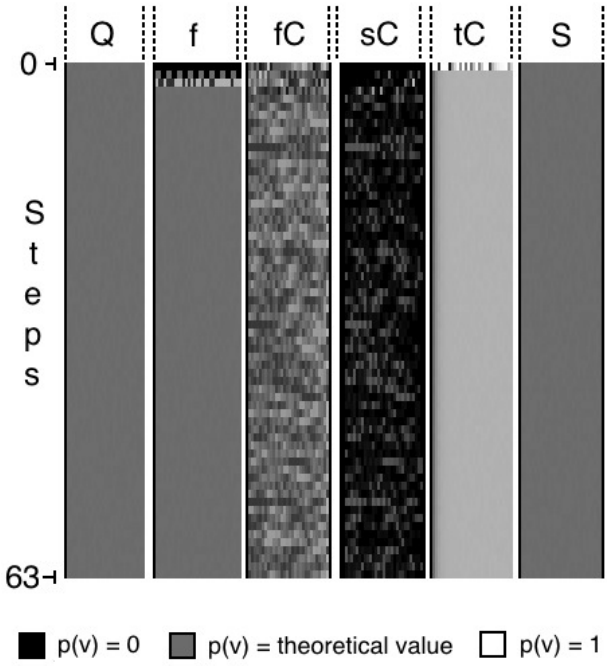

Figure 14: Figure representing a differential vector of probabilities on variables from the MD5 process, such as each pixel is about $P(v|v-1)$ to be 1, from the MD5 process, step 0 to 63, sorted by type of 32-bits word on big endian.

5.4 Gather probabilistic and judicious information from MD5

When such a statistical database is built, specific treatments can be applied to refine the knowledge associated to the problem. In our works, that's what we did by searching for specific relations which have a very high probability to exist. In practice, several factual relations were detected, as for instance fixed variables and equivalences but also probabilistic relations as quasi-fixed variables, quasi-implications or quasi-equivalences.

5.4.1 Factual relations

We call *factual relation* a relation connecting two variables that occurs no matter the input message. In fact, this corresponds to the probabilities which are turned to 1 or 0. In such cases, fixed variables are identified. Furthermore, it's also possible to detect implications and so equivalencies. Indeed, let be v and w two boolean variables:

1) $if\ p(v) = 0\ (resp 1),\ then\ v = 0\ (resp 1)$

2) $if\ p(v) = p(v \wedge w),\ then\ v \Rightarrow w$. In others words: $if\ p(w|v) = 1, then\ v \Rightarrow w$

Relation	From MD5
Quasi-Equivalences	$\overline{Q_1[21]} \nLeftrightarrow M_0[14], Q_{61}[8] \nLeftrightarrow Ha[8], Q_{61}[8] \nLeftrightarrow cHa[8]$ $Ha[8] \nLeftrightarrow Q_{61}[8], Ha[8] \nLeftrightarrow cHa[8], cHa[8] \nLeftrightarrow Q_{61}[8], M_0[14] \nLeftrightarrow f_1[21]$ $cHa[8] \nLeftrightarrow Ha[8], f_1[21] \nLeftrightarrow M_0[14], M_0[14] \nLeftrightarrow Q_1[21]$
Quasi-Implications	$M_0[14] \nRightarrow Q_1[21], Q_1[21] \nRightarrow M_0[14], Ha[8] \nRightarrow Q_{16}[8]$ $Q_{61}[8] \nRightarrow Ha[8], cHa[8] \nRightarrow Q_{61}[8], Q_{64}[28] \nRightarrow cHb[30]$ $Q_{61}[8] \nRightarrow Ha[8], Ha[8] \nRightarrow Q_{61}[8], cHa[8] \nRightarrow Ha[8]$ $Hb[28] \nRightarrow cHb[30], cHa[8] \nRightarrow Q_{61}[8], Ha[8] \nRightarrow cHa[8]$
Quasi-Fixed variables	$cHa[7] \simeq 0$ *where* $p(cHa[7]) = 0.0040$ $cHc[7] \simeq 1$ *where* $p(cHc[7]) = 0.9923$ $sC_0[20] \simeq 0$ *where* $p(sC_0[20]) = 0.0081$ $sC_1[26] \simeq 0$ *where* $p(sC_1[26]) = 0.0044$
Relation	From SHA-1
Quasi-Equivalences	$sC_0[28] \nLeftrightarrow sC_0[17], sC_0[27] \nLeftrightarrow fC_0[28], sC_0[24] \nLeftrightarrow cHc[7], sC_0[24] \nLeftrightarrow fC_0[28]$ $sC_0[17] \nLeftrightarrow cHc[7], sC_0[17] \nLeftrightarrow fC_0[28], fC_1[16] \nLeftrightarrow sC_1[16], sC_1[16] \nLeftrightarrow fC_1[16]$
Quasi-Implications	$\overline{M_0[8]} \nRightarrow sC_1[14], M_1[13] \nRightarrow sC_1[14], M_1[10] \nRightarrow sC_1[14]$ $M_2[25] \nRightarrow sC_2[27], M_3[27] \nRightarrow sC_3[27]$
Quasi-Fixed variables	$cHa[6] \simeq 0$ *where* $p(cHa[6]) = 0.0082$ $cHc[7] \simeq 1$ *where* $p(cHc[7]) = 0.9922$ $fC_0[28] \simeq 1$ *where* $p(fC_0[28]) = 0.9904$ $sC_0[28] \simeq 0$ *where* $p(sC_0[28]) = 0.0096$

Table 4: Some of Quasi-Equivalences and Quasi-implications detected from the MD5 and the SHA-1 processes with a threshold $t = 0.99$. The used notation is the one presented in 2.2. Moreover Hx and cHx are variables about the hash.

3) *if* $p(v|w) = p(w|v) = 1$, *then* $v \Leftrightarrow w$

5.4.2 Probabilistic relations

A *probabilistic relation* is a relation connecting two variables that exists in most cases. Obviously, this kind of relation depends of a threshold which determine if the relation has a good chance to exist or not. When such a relation is used, the threshold corresponds to a lower bound above which the probability of the relation is. At last, this section also deals with *quasi-relation* to mentioned a probabilistic relation.

Formally, let be t a threshold such as $t \leq p(v) < 1$. We call *quasi-fixed variable* a variable such as whatever the instance, the variable has a probability $\mathrm{p}(v)$ to be 1 higher than t. This means, the variable v (resp $\overline{v}$) is set to 1 (resp 0) in ($\mathrm{p}(v)$*100) % of cases.

We call *quasi-implication* a relation between two variables such as:

$$\mathrm{p}(w|v) \geq t \text{ and note this relation } v \nRightarrow w$$

Finally, we call *quasi-equivalence* a relation between two variables such as:

$$\mathrm{p}(w|v) \geq t,\ \mathrm{p}(v|w) \geq t \text{ and note this relation } v \nLeftrightarrow w$$

5.4.3 Application to MD5 and SHA-1

In this part, we just focus on the detection of probabilistic relations, because detecting factual relations is already done by the previous defined logical preprocessing (section 4).

In practice, a specific treatment was applied to extract all the specific probabilities from that quasi-relations can be concluded. For instance, by defining a threshold $t = 0.99$, we found $\mathrm{p}(\overline{M_0[14]} \mid f_2[21]) = 0.9969$ from the MD5 process. This means $f_2[21] \nRightarrow \overline{M_0[14]}$ and this can be added to the formula by the new clause $(\overline{f_2[21]} \vee \overline{M_0[14]})$.

We concretely experimented this framework to MD5 and SHA-1 by giving a portion of different relations we detected in the table 4. Note that a lot of relations are in the first steps and could facilitate the prediction of the probabilistic behavior of the invoked variables. Furthermore, if many relations between carries are present there are also other ones involving internal states, the input message or non-linear functions. In our knowledge, this is the only approach where computed probabilities are concretely put together in order to emerge new relations that could be exploit in practice.

To put the stress on the interest of our experimentations, are referenced in the table 5 the number of equivalences, quasi-fixed variables, quasi-equivalences and quasi-implications found from the SAT formulas representing MD5 and SHA-1, according to a threshold t. It can be seen that more the threshold is low, more information are got about special relations between variables. Note also that the number of quasi-implication does not follow this growth. This is explain by the fact that quasi-implications are correlated with quasi-fixed variables. Indeed, if a variable v_k is not quasi-fixed and have a very high probability to be 1

MD5			
Threshold	q-Fixed	q-Impli	q-Equi
0.995	2	638	10
0.99	5	2089	29
0.985	9	21995	79
0.98	12	2048	105
SHA-1			
Threshold	q-Fixed	q-Impli	q-Equi
0.995	1	75	8
0.99	5	223	44
0.985	8	5244	88
0.98	13	419	169

Table 5: Detection of quasi-fixed variables (q-Fixed), quasi-implications (q-Impli) and quasi-equivalences (q-Equi) in the MD5 and SHA-1 process according to a threshold

(near from the threshold), it may exists a lot of relations such as $v_i \nRightarrow v_k$, where $v_i \in \mathcal{V}$.

Finally, we think detecting probabilistic relations are a good way to enrich again our SAT formulas by adding information that are nonexistent otherwise. In the next section, different other ways are mentioned to exploit these *quasi-relations* within a practical framework.

6 How to use *quasi-relations*

Practical attacks in logical cryptanalysis offer a perfect framework to exploit weaknesses targeted on some bits. In the context of our works, all the *quasi-relations* that were detected provide a set of information that could be exploited in searching preimages for cryptographic hash functions. In this manner, the practical complexity of a preimage problem can be reduced during the boolean encoding or directly during the solving phase (see part **??**).

As several *quasi-fixed* variables were found, the practical complexity of a problem can be reduced by assigning a value to the corresponding variables before the solving phase. Indeed, for instance if a variable v has a very high probability to be set to 1, it can be decided to concretely fix its value to 1. By doing so, all the clauses containing the corresponding positive literals v are SAT and all the clauses containing the corresponding negative literals $\overline{v}$ are reduced. Moreover, the found *quasi-equivalences* can be used to replace a variable by an other one. Doing this allows to provoke many simplifications as for instance the ones presented in this paper in section 4.

Even if these approaches seems to be easily exploitable, it should be remembered that

quasi-relations are probabilistic and not always verified. Consequently, all of these probabilistic simplifications are beneficial but present a certain risk. Indeed, to do so implies assigning and determining a part of the solution and consequently cutting a piece of the search space. Despite this fact, this is a measured bet in the sense that making a choice, as for instance setting a value to a variable allows to prune a branch of the binary search tree during the solving phase. Nevertheless, this pruning divides by two the search space while the solution space is only affected by the probability of the *quasi-fixed* variable. Indeed, when a *quasi-relation q* is used, the search space decreased about a factor (1 - $p(q)$). Next, we are interesting in counting how many of *quasi-relations* could be used in order to help the solving of a problem.

Let be a detection process of *quasi-relations* where the threshold t equals to 0.995. Because this computation is theoretical, we consider the worst case where all the *quasi-relations* are found with a probability equals to t, and no more. Thereby, the principle is as following: if one *quasi-relation* is used, the search space decreases to 99.5% of its original size. If two *quasi-relations* are imported, the space search decrease about a factor $t^2 = 0.995^2$... and so on. In a practical framework, if a cryptanalyst considers that a good search space can decrease to 50%, he can use *n quasi-relations* where:

$$n = \frac{log(\frac{1}{2})}{log(0.995)} = 138$$

This number shows that the use of *quasi-relations* is rather quickly limited. However, probabilities are not equitably distributed in a real case. The tool we implemented is able to detect *quasi-relations* by extracting them one by one, by privileging the highest probabilities in first while a confidence of nearly 50% is ensured. Within a practical framework, once the best quasi-relations uprooted, they were injected in the CNF formulas. Then a preprocessing was applied in order to reduce the size of the SAT formula. The table 6 gives the new statistics of our CNF according to t. The column *Conf* (for Confidence) shows the percentage of how the search space has narrowed.

In this manner, adding probabilistic information allows to decrease the sizes of our SAT formulas and thus their practical complexity while maintaining a good solution space.

7 Practical attacks against cryptographic hash functions

Up to now, this paper deals with how to model a hashing process into a SAT formalism and how to use the resulting logical formula in order to have a new angle of view of the problem. From this, we conclude by an improvement of the structural understanding of the modeled problem and the discover of some weaknesses. In this way, we show that SAT can be used to get detailed information about the variables, and so the internal bits of the

MD4				
Experimentation	**Clauses**	**Variables**	**Literals**	**Conf**
Without proba.	171 235	12721	≈814 096	100%
With proba.	171 028	12705	≈ 813 343	52.3%
MD5				
Experimentation	**Clauses**	**Variables**	**Literals**	**Conf**
Without proba.	171 235	12721	≈814 096	100%
With proba.	171 028	12705	≈ 813 343	49.3%
SHA-1				
Experimentation	**Clauses**	**Variables**	**Literals**	**Conf**
Without proba.	375 195	12 771	$\approx 2.21 \cdot 10^6$	100%
With proba.	374 541	12 732	$\approx 2.18 \cdot 10^6$	52%

Table 6: Reduction of the SAT formulas sizes for MD5 and SHA-1 thanks to a probabilistic providing. The new instances keep at most 50% of their solution space.

process. In this section, the point of view had changed and is focused on practical attacks. To illustrate a concrete use of our formulas, this part highlights how to tackle the preimage problem and in what way SAT can help this attack. More precisely, details and results about preimage attacks are given and some experimentations underline both our contribution and our methodology.

7.1 About preimage attacks

In order to tackle the preimage, a first work consists in the generation of a SAT formula representing the hash function. Then in a second step, the variables corresponding to a digest are set. From this, it results a new SAT formula. Solving such an instance amounts to saying that the solver find an assignment to each variable allowing to satisfy all the constraints. This means, the variables corresponding to the input message are also defined, and so an input message can be reconstructed. In this way, this answers to the preimage problem. In fact, solving such an instance is the same as inverse the one-way cryptographic hash function. However, even if this prospect is a treasure coveted by cryptanalysts, it remains fortunately too difficult to be achieved in practice. It's why, an intuitive choice to measure the resistance of cryptographic hash function thanks to logical cryptanalysis is to consider an incremental approach by studying reduced-step process SAT formulas.

We put into practice this approach. In a first phase, reduced-step formulas with a fixed digest were generated, from a first nontrivial problem to the whole hashing process. For instance, for the MD5 hash function all the SAT formulas from the step 17 to the step 64 were

made. Then, in a second phase, these formulas were preprocessed and solved, beginning by the easiest CNF to the furthest away possible. The last formula to be solved is the practical limit of the dedicated attack.

Within this context, it is easy to lead some experimentations about how SAT solvers are efficient to solve our formulas relatively to the diverses parameters of a SAT formula: the number of step modeled, the repercussions of a prior preprocessing or of an adding of probabilistic information.

7.2 Improved runtimes thanks to SAT

In this part we present some results that show how SAT can help to accelerate a solving. First we present runtimes of SAT solving of formulas that corresponds to preimage attacks on reduced-step process for the MD5 primitive. Second, we show the practical consequences of an import of probabilistic information into our CNF formulas relatively to the performance a SAT solving. Both the experimentations have been done using the parallel SAT solver PLINGELING[47] applied on a two processors Intel Ivy Bridge 8 cores 2,6 GHz[10].

In the tab 7, we repertories runtimes on average about at least 100 solving of each formula relative to the different modeling and its logical simplification. Here, we can easily see that all the runtimes obtained on a SAT formula that have been preprocessed is better, that it is not. This means that purify a SAT formula is efficient to improve a solving. In addition, we also compared the runtimes obtained on our SAT formulas (on which) a preprocessing was applied and SAT formulas with a probabilistic information. One more time, we see that the runtimes decrease thanks to the probabilistic import.

This experimentation shows that an upstream work on the modeling phase can really be helpful in order to improve practical attacks.

7.3 Best practical attacks

The previous presented works show experimentations about practical attacks on reduced step formulas. In order to get the best practical attack, we tested how many steps we can inverse thank to SAT solving. Thereafter in the table 8 a comparison between, in our knowledge, the best practical attacks against the full preimage of hash functions we tackled. These ones are mainly based on logical cryptanalysis because this domain is perfectly adapted to this approach.

From a technical point of view, our best results were obtained thanks to a parallel complete SAT solver called PLINGELING applied on a a two processors Intel Ivy Bridge 8 cores 2,6 GHz. We choose to use this SAT solver for diverse reasons as for instance the fact it exploits parallelism that could help to improve our results on a multi-cores machine. Moreover, in practice many solvers were tested to determine which is the most appropriate to

[10]Thanks to the *ROMEO HPC Center in Champagne-Ardenne*, https://romeo.univ-reims.fr

SAT SOLVER			PLINGELING
FORMULA	SIMPLIFIED?	PROBAS ADDED?	TIME (S)
md5-25-steps	no	no	3.8
md5-25-steps	yes	no	2.3
md5-25-steps	yes	yes	**1.8**
md5-26-steps	no	no	8.3
md5-26-steps	yes	no	5.4
md5-26-steps	yes	yes	**4.0**
md5-27-steps	no	no	8.8
md5-27-steps	yes	no	5.2
md5-27-steps	yes	yes	**4.1**

Table 7: Runtimes on average about at least 100 solving of each formula relative to the modeled primitive, its logical simplification and a probabilistic import in the CNF.

	Best practical attacks	**In this paper**
MD4	39 steps in [22] [48]	39 steps
MD5	26 steps in [22]	**28 steps**
SHA-0	23 steps in [49] [50]	23 steps
SHA-1	23 steps in [49] [50]	23 steps

Table 8: Best practical attacks on step-reduced MD[4/5] and SHA-[0/1] preimages

tackle our instances but in reality the best of them are quiet equivalent on these specific cryptographic problems. Chosing one or another doesn't impact the quality of the results.

Hereafter, some results input/output values, in big endian, where the digests correspond to the ones obtained after the last addition of the primitive.

2 rounds 7 steps on MD4 (39 steps) [11]

Fixed Hash:

```
0x00000000 0x00000000 0x00000000 0x00000000
```

Input found:

```
0x321838cd 0x67867da5 0x67867da5 0x4bd844ff 0x67867da5 0x67867da5 0x67867da5 0x60babe30
0x67867da5 0x67867da5 0x67867da5 0x2e731890 0xb84655eb 0x1094c071 0xce0cfe36 0x0252233c
```

[11] With a classical approach, MD4's limit was about 31 steps. But this limit could had been improved by the import of the Dobbertin's attack[23]

1 round 12 steps on MD5 **(28 steps)**

Fixed Hash:
0x01234567 0x89abcdef 0xfedcba98 0x76543210
Input found:
0x0bd86c16 0x6dea158a 0x3fea904c 0x5930a4a1 0xf733709c 0x7e818951 0xdc6f481b 0x21f85c42
0x7a6b2051 0x09762af5 0xbf21286b 0xb70fe9bc 0xb6e76e81 0x0ba31a2c 0x71512697 0xbc2931af

1 round 3 steps on SHA-0 **(23 steps)**

Fixed hash:
0x00000000 0x00000000 0x00000000 0x00000000
Input found:
0x4e073784 0x050bf14c 0xa5d325cb 0xf81d3ce2 0x123bd6f4 0xe71fc07f 0xf07fda73 0x43b05533
0x704efd95 0xc8178c49 0x02c891de 0x1f7630ae 0x4130f392 0x55113db5 0xacc0022f 0x3b8c154d

1 round 3 steps on SHA-1 **(23 steps)**

Fixed hash:
0x00000000 0x00000000 0x00000000 0x00000000
Input found:
0x35691c1a 0xead7eb26 0xcac76b0e 0x00000000 0x51e43c45 0xaa8bc12a 0xdb8fa47c 0x00000000
0x637c1517 0x80abea2e 0x9339f44e 0x00000000 0x6367caee 0xbc8920ec 0x1084c8d7 0x45075a9e

7.4 Preimage attacks: analysis

By looking at the table 1, more steps are inverted in our works for the MD5 hash function than the previous attacks in the literature. Concerning SHA-[0/1] the presented results are equivalent to those of [49]. Regarding this last work, it can be remarked that their way to model hash functions is quiet the same as ours, *i.e.* their formulas are handcrafted. This is not the case for the works presented in [22] were the authors make reference of an automatic tool (generally based on *Tseitin* transformations) to generate their CNF.

On this last point, a question that could cross our minds is the fact that new SAT solvers may just be powerful than the one used in previous works. To answers this, we make the same experimentation by using MINISAT V2.2 as in [22] and we inverted more steps whether for the MD4 (without an import of the Dobbertin's attack) and the MD5 hash function. We explain this by a more smooth modeling of our CNF formulas. For instance, the SAT formula representing 28 steps of MD4 is composed of 54,508 clauses and 4,522 variables while the one of [22] contains 202,407 clauses and 3,007 variables. In our point of view this point underlines the importance of a SAT expertise to precisely measure the resistance of such cryptographic hash functions.

To be more complete about our preimage attacks, we are also interested in the evolution of the runtime relative to the number of steps modeled. These observations, except the last level, were achieved on an average of 25 solving and drawn in the figure 15, with a logarithmic scale. Our benchmarks have been achieved on a MacBookPro 1.7 GHz thanks to a sequential SAT-solver called LINGELING [47]. Here the parametrization of these experimentations was chosen just by considering a working environment without too many perturbations.

It can be observed that the runtimes keep the same evolution whatever the modeled hash function, following an exponential growth according to the number of steps modeled. The MD4 case is special due to the import of the Dobbertin's attack [23] to improve the attack (Details are given in [48]). In this manner, these hash functions can be considered *a priori* relatively secure against preimage attacks by SAT solvers. However, the solving method is near from a cleaver brute-force attack in that SAT solvers mixes the best techniques from *A.I* but are not specified to this kind of problem. By the way, in our owns experimentations whatever the used SAT solver, the runtimes are quasi the same.

8 Conclusion

The works presented in this paper show a practical application of logical cryptanalysis about cryptographic hash functions. The first part is dedicated to the modeling of such a primitive into a SAT formalism and presents many simplifications that are possible thanks to the binary context of SAT. In this way, we underlined the interest of this particular algebraic cryptanalysis to purify the concrete representation of a problem through an efficient and adapted preprocessing. Furthermore, this have also allowed to the discover of logical inferences and so simplifications in order to reduce the computational complexity of the SAT solving. We highlight that these relations could also be exploited in an other cryptanalytic framework as for instance differential attacks. A second part of this paper is dedicated to a probabilistic reasoning. From this, a weakness about the use of a round constant at each step is illustrated thanks to pictures that compares expected and computed probabilities and by the discover of new probabilistic relations, called quasi-relations. These relations are important because they put the stress on new information that could be used for both smooth the modeling of problems and increase the efficiency of SAT solving. Added to that, this approach seems to be the first one in a logical cryptanalysis framework to utilize a SAT formula in order to find a weakness in a cryptographic primitive. In our point of view, this is an hopeful sign to have a more interested view on logical cryptanalysis. Finally, a last part deals with practical attacks about a search of preimage. In this manner, we present some results about the MD[4/5] and SHA-[0/1] families cryptographic hash functions that are at least as good as the ones from the literature. This attests a certain interest of these works and open some prospects.

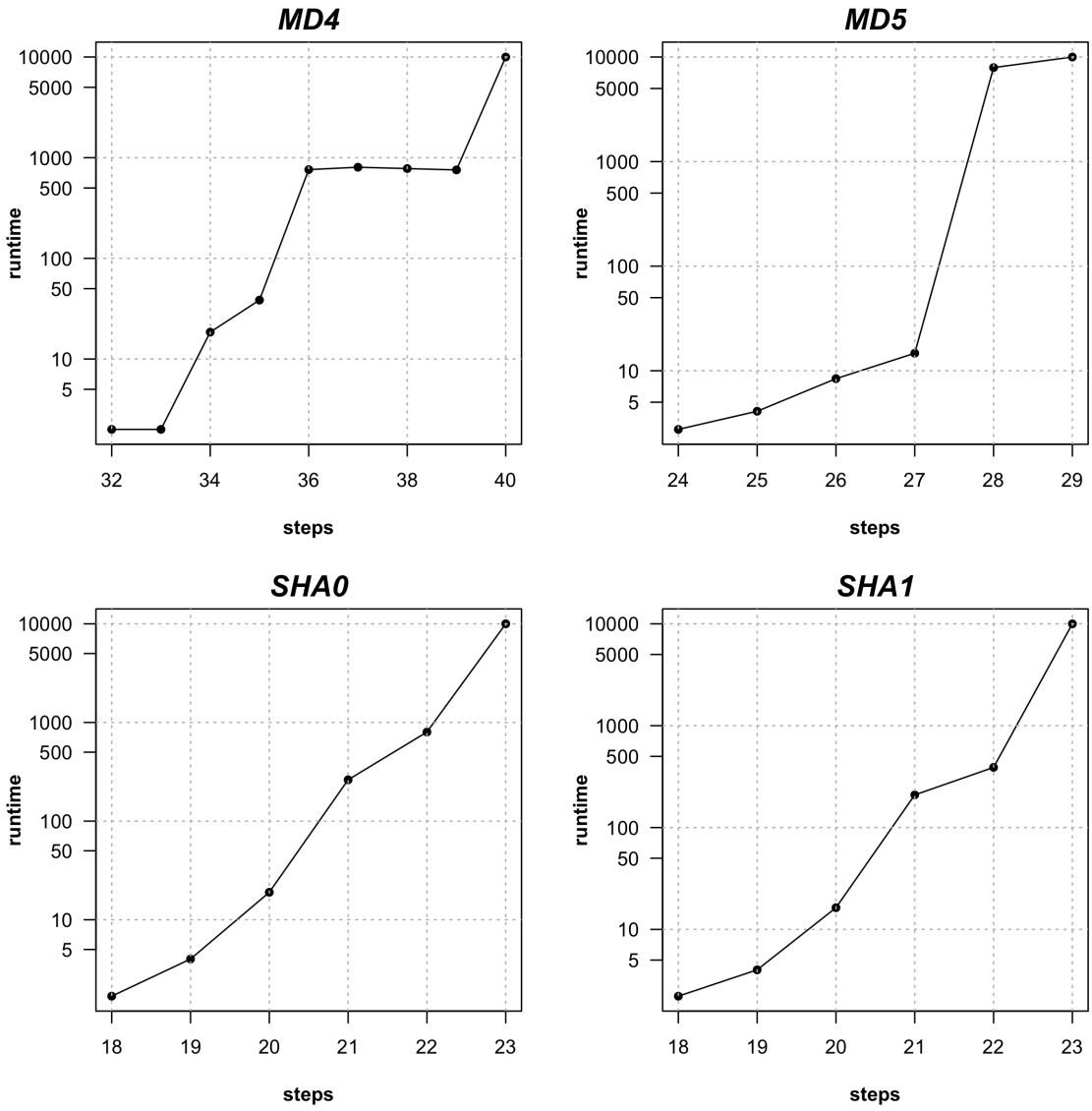

Figure 15: Runtimes for solving reduced-step instances representing MD4, MD5, SHA-0 and SHA-1, depending on the number of steps tackled

Indeed, logical cryptanalysis is a new way to approach a problem and the SAT expertise could be more exploited. One of the main idea is to specify a SAT solver to tackle cryptographic problems. A good way to solve algebraic systems, especially boolean systems, is the use of *complete* SAT solvers. These ones are mainly based on the DPLL [2] and/or

the CDCL [51] algorithms which consist in a systematic enumeration of truth assignments thanks to a binary search-tree (see section 2). Each node of this search-tree represents a current assignment where a policy choice is made to decide the next variable to be assign. A probabilistic providing can be a good mean to improve the splitting choice policy of a dedicated SAT solver. In practice, the heuristic branching computes a very precise evaluation to determine the new node of the search tree, but this choice is often difficult. Here, this evaluation can be helped by the adding of a probabilistic knowledge. Added to this, a better understanding of the structures that are hidden in the SAT formulas may bring new information about the modeled cryptographic primitive, as for instance the presence of clusters or the definition of satisfaction profiles for each clauses. Finally, it could be interesting to export the whole methodology to others cryptographic functions where logical cryptanalysis may be even more effective, as for instance lightweighted cryptography.

References

[1] S. A. Cook. The Complexity of Theorem Proving Procedures. *In 3rd ACM Symp. on Theory of Computing, Ohio*, pages 151–158, 1971.

[2] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem-Proving. *Journal Association for Computing Machine*, (5):394–397, 1962.

[3] F. Bacchus and J. Winter. Effective preprocessing with hyper-resolution and equality reduction. 2003.

[4] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *Proc. of ICCAD*, San Jose, Nov 2001.

[5] H. Kautz and B. Selman. Pushing the Envelope: Planning, Propositional Logic and Stochastic Search. In *Proc. of the 30th National Conf. on Artificial Intelligence and the 8th Innovative Applications of Artificial Intelligence Conf.*, pages 1194–1201, Menlo Park, August4–8 1996. AAAI Press / MIT Press.

[6] Armin Biere, Keijo Heljanko, Tommi Junttila, Timo Latvala, and Viktor Schuppan. Linear encodings of bounded LTL model checking. *Logical Methods in Computer Science*, 2, 2006.

[7] Nachiketh R. Potlapally, Anand Raghunathan, Srivaths Ravi, Niraj K. Jha, and Ruby B. Lee. Aiding side-channel attacks on cryptographic software with satisfiability-based analysis. *IEEE Trans. VLSI Syst.*, 15(4):465–470, 2007.

[8] M. Matsui and A. Yamagishi. A new method for known plaintext attack of feal cipher. In *EUROCRYPT*, pages 81–91, 1992.

[9] E. Biham and A. Shamir. Differential cryptanalysis of des-like cryptosystems. In *CRYPTO*, pages 2–21, 1990.

[10] Diffie and Hellman. Exhaustive cryptanalysis of the nbs data encryption standard. In *Computer 10 (6)*, pages 74–84, June 1977.

[11] Ambrose and al. *Power Analysis Side Channel Attacks: The Processor Design-level Context*. VDM Verlag, 2010.

[12] F. Massacci and L. Marraro. Logical cryptanalysis as a sat problem. *J.Autom.Reasoning*, pages 165–203, 2000.

[13] Henry Yuen Joseph Bebel. Hard sat instances based on factoring. In *Proceedings of the SAT competition 2013*, page 102, 2013.

[14] Vegard Nossum. Instance generator for encoding preimage, second-preimage, and collision attacks on sha-1. In *Proceedings of the SAT competition 2013*, pages 119–120, 2013.

[15] Mate Soos. Grain of salt benchmarks. In *Proceedings of the SAT competition 2013*, page 121, 2013.

[16] I. Mironov and L. Zhang. Applications of sat solvers to cryptanalysis of hash functions. In *SAT*, pages 102–115, 2006.

[17] Tobias Eibach, Enrico Pilz, and Gunnar Völkel. Attacking bivium using sat solvers. In *SAT*, pages 63–76, 2008.

[18] Mate Soos. Enhanced gaussian elimination in dpll-based sat solvers. In Daniel Le Berre, editor, *POS-10*, volume 8 of *EPiC Series*, pages 2–14, 2010.

[19] Nicolas T. Courtois. Algebraic complexity reduction and cryptanalysis of gost, 2011.

[20] Constantinos Patsakis. Rsa private key reconstruction from random bits using sat solvers. *IACR Cryptology ePrint Archive*, 2013:26, 2013.

[21] Ekawat Homsirikamol, Pawel Morawiecki, Marcin Rogawski, and Marian Srebrny. Security margin evaluation of sha-3 contest finalists through sat-based attacks. In *CISIM*, pages 56–67, 2012.

[22] D. De, A. Kumarasubramanian, and R. Venkatesan. Inversion attacks on secure hash functions using satsolvers. In *SAT*, pages 377–382, 2007.

[23] H. Dobbertin. Cryptanalysis of md4. In *FSE*, pages 53–69, 1996.

[24] G. S. Tseitin. On the complexity of derivation in propositional calculus. In A. O. Slisenko, editor, *Studies in Constructive Mathematics and Mathematical Logic II*, volume 8 of *Seminars in Mathematics: Steklov Mathem. Inst.*, pages 115–125. Consultants Bureau, New York, 1970.

[25] R. Merkle. One way hash functions and des. In *CRYPTO*, pages 428–446, 1989.

[26] I. Damgard. A design principle for hash functions. In *CRYPTO*, pages 416–427, 1989.

[27] X. Wang. Collisions for hash functions md4, md5, haval-128 and ripemd. In *Crypto'04*, page 199, 1997.

[28] X. Wang and H. Yu. How to break md5 and other hash functions. In *EUROCRYPT*, pages 19–35, 2005.

[29] H. Yu and X. Wang. Multi-collision attack on the compression functions of md4 and 3-pass haval. In *ICISC*, pages 206–226, 2007.

[30] Christophe De Cannière, Florian Mendel, and Christian Rechberger. Collisions for 70-step sha-1: On the full cost of collision search. In *Selected Areas in Cryptography*, pages 56–73, 2007.

[31] Somitra Kumar Sanadhya and Palash Sarkar. New collision attacks against up to 24-step sha-2. In *INDOCRYPT*, pages 91–103, 2008.

[32] Marc Stevens, Arjen K. Lenstra, and Benne de Weger. Chosen-prefix collisions for md5 and applications. *IJACT*, 2(4):322–359, 2012.

[33] Tao Xie and Dengguo Feng. How to find weak input differences for md5 collision attacks. *IACR Cryptology ePrint Archive*, 2009:223, 2009.

[34] Yu Sasaki and Kazumaro Aoki. Finding preimages in full md5 faster than exhaustive search. In *EUROCRYPT*, pages 134–152, 2009.

[35] Marc Stevens. New collision attacks on sha-1 based on optimal joint local-collision analysis. In *EUROCRYPT*, pages 245–261, 2013.

[36] F.Chabaud and A. Joux. Differential collisions in sha-0. In *CRYPTO*, pages 56–71, 1998.

[37] Gilles Audemard and Laurent Simon. Gunsat : a greedy local search algorithm for unsatisfiability. In *International Joint Conference on Artificial Intelligence(IJCAI'07)*, pages 2256–2261, jan 2007.

[38] A. Biere, M. J. H. Heule, H. Van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.

[39] Shaowei Cai and Kaile Su. Configuration checking with aspiration in local search for sat. In *AAAI*, 2012.

[40] Dave A. D. Tompkins, Adrian Balint, and Holger H. Hoos. Captain jack: New variable selection heuristics in local search for sat. In *SAT*, pages 302–316, 2011.

[41] Jo ao P. Marques Silva and Karem A. Sakallah. Grasp a new search algorithm for satisfiability. In *ICCAD '96: Proc. of the 1996 IEEE/ACM Intern. Conf. on Computer-aided design*, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[42] Jo ao P. Marques Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning sat solvers. In *Handbook of Satisfiability*, pages 131–153. 2009.

[43] Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In *SAT*, pages 61–75, 2005.

[44] F. Bacchus and J. Winter. Effective preprocessing with hyper-resolution and equality reduction. In *SAT*, 2003.

[45] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *IJCAI (1)*, pages 366–371, 1997.

[46] Sylvain Darras, Gilles Dequen, Laure Devendeville, Bertrand Mazure, Richard Ostrowski, and Lakhdar Sais. Using boolean constraint propagation for sub-clauses deduction. In *CP*, pages 757–761, 2005.

[47] A. Biere. Lingeling, plingeling, picosat and precosat at sat race 2010. In *Tech. Rep. 10/1, FMV Reports Series, Johannes Kepler University, Altenbergerstr. Linz, Austria*, pages 244–257, 2010.

[48] Florian Legendre, Gilles Dequen, and Michaël Krajecki. Inverting thanks to sat solving - an application on reduced-step md*. In *SECRYPT*, pages 339–344, 2012.

[49] Vegard Nossum. Sat-based preimage attacks on sha-1. *Thesis*, 2012.

[50] Florian Legendre, Gilles Dequen, and Michaël Krajecki. Encoding hash functions as a sat problem. In *ICTAI*, pages 916–921, 2012.

[51] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD*, 2001.