Logical Analysis of Hash Functions

Dejan Jovanović[1] and Predrag Janičić[2]

[1] Mathematical Institute,
Kneza Mihaila 35, 11000 Belgrade, Serbia and Montenegro
dejan@mi.sanu.ac.yu
[2] Faculty of Mathematics,
Studentski trg 16, 11000 Belgrade, Serbia and Montenegro
janicic@matf.bg.ac.yu

Abstract. In this paper we report on a novel approach for uniform encoding of hash functions (but also other cryptographic functions) into propositional logic formulae, and reducing cryptanalysis problems to the satisfiability problem. The approach is general, elegant, and does not require any human expertise on the construction of a specific cryptographic function. By using this approach, we developed a technique for generating *hard and satisfiable* propositional formulae and *hard and unsatisfiable* propositional formulae. In addition, one can finely tune the hardness of generated formulae. This can be very important for different applications, including testing (complete or incomplete) SAT solvers. The uniform logical analysis of cryptographic functions can be used for comparison between different functions and can expose weaknesses of some of them (as shown for MD4 in comparison with MD5).

1 Introduction

Hash functions have wide and important role in cryptography. They produce hash values, which concisely represent longer messages or documents from which they were computed. Examples of hash functions are MD4, MD5, and SHA. The main role of cryptographic hash functions is in the provision of message integrity checks and digital signatures.

The subject of research presented in this paper is analysis of hash functions in terms of propositional reasoning.[1] We will try to shed a new light on hash functions and to address several important issues concerning hash functions and SAT problem. First question considered is whether the problem of inverting a hash function can be reduced to a SAT problem; if yes, how it can be done effectively. Section 4.1 gives one methodology for this.

Another question of interest is: *can hash functions be used for generating hard instances of* SAT *problem?* The need for hard instances of SAT problem is well-explained in [1]:

[1] The work presented in this paper is, in a sense, parallel to [12], as it investigates hash functions in a similar manner the work reported in [12] investigates DES algorithm.

> “A major difficulty in evaluating incomplete local search style algorithms for constraint satisfaction problems is the need for a source of hard problem instances that are guaranteed to be satisfiable. A standard approach to evaluate incomplete search methods has been to use a general problem generator and a complete search method to filter out the unsatisfiable instances. Unfortunately, this approach cannot be used to create problem instances that are beyond the reach of complete search methods. So far, it has proven to be surprisingly difficult to develop a direct generator for satisfiable instances only.“

In [1], it is claimed that cryptographic algorithms cannot be used for generating interesting hard SAT instances, as the problems are too hard (require exhaustive search) and cannot be fine-grained. In this paper, we question these claims and show how hash functions can be used for generating satisfiable SAT instances of finely tuned hardness. We will also consider generating unsatisfiable SAT instances. Namely, while satisfiable instances are required for testing completeness, unsatisfiable instances are required for testing soundness[2] of complete SAT solvers.

Why it is difficult to randomly generate hard satisfiable instances of SAT problem is also discussed in [8]. A survey [4] points that generating hard *solved* instances of SAT problem is equivalent to computing an one-way function, which in turn is equivalent to generating pseudo-random numbers and private-key cryptography. The work [4] also discusses how a fixed-length one-way function can be used to generate hard solved instances of 3SAT. We are not aware that this proposal has been used in practice and it seems that it would be very difficult to apply it to real-world hash functions. We believe that the approach we present in this paper is more elegant and applicable to state-of-the-art hash functions, provided their implementations. In addition, our approach can be used for producing both *satisfiable* SAT *instances of finely tuned hardness* and *unsatisfiable* SAT *instances of finely tuned hardness.*

It is interesting whether logical analysis could expose weaknesses of some hash functions. Finding such a weakness often relies on a human expertise and is most often specific for a certain sort of problems. Therefore, it would be good if a (uniform) logical analysis could provide a deeper understanding of nature of hash functions and expose their potential weaknesses. Experimental results, given in §6, based on uniform logical analysis, show that MD4 function is much weaker than MD5 (as expected).

Another very interesting question is whether such SAT formulae are the hardest SAT formulae (within the class of formulae with the same number of variables). We will briefly comment on this question and possible ways for investigating it within our plans for future work.

[2] Since SAT solvers are becoming a standard tool in many critical industrial applications, testing the solver for soundness is of uttermost importance.

2 Background

Hash Functions. A hash function *hash* is a transformation that takes an input sequence of bits m (the message) and returns a fixed-size string, which is called the *hash value* (also the *message digest*, the *digital fingerprint*). The basic requirement for a cryptographic hash function is that the hash value does not reveal any information about the message itself, and moreover that it is hard to find other messages that produce the same hash value. If only a single bit of the message is changed, it is expected that the new hash value is dramatically different from the original one. A hash function is required to have the following features:

Preimage resistant. A hash function *hash* is said to be *preimage resistant* if it is hard to invert, where "hard to invert" means that given a hash value h, it is computationally infeasible to find some input x such that $hash(x) = h$.

Second preimage resistant. If, given a message x, it is computationally infeasible to find a message y different from x such that $hash(x) = hash(y)$, then *hash* is said to be *second preimage resistant.*

Collision-resistant. A hash function *hash* is said to be *collision-resistant* if it is computationally infeasible to find two distinct messages x and y such that $hash(x) = hash(y)$.

A hash function must be able to produce a fixed-length output for an arbitrary-length message. This is usually achieved by breaking the input into series of equal-sized blocks, and then operating on these blocks in a sequence of steps, using compression functions. Often, the last block processed also contains the message length, which improves the properties of the hash. This construction is known as the Merkle-Demagård structure [5,13], and the majority of hash functions in use are of this form, including MD4, MD5 and the SHA family.

MD4 and MD5 are message-digest algorithms developed by Ron Rivest [16,17]. These two algorithms take a message of arbitrary length and produce a 128-bit message digest. Attacks on versions of MD4 with either the first or the last rounds missing were developed very quickly. Also, it was shown how collisions for the full version of MD4 can be found in under a minute on a typical PC. MD5 algorithm is basically an improved version of MD4. The algorithm consists of four distinct rounds, which have a slightly different design from that of MD4. Collisions for the full MD5 were announced in 2004 [19], and the attack was reported to take only one hour on an IBM P690 cluster. This year it was demonstrated that using the methodology of previous attacks it is possible to construct two X.509 certificates with different public keys and the same MD5 hash value [11]. However, this still does not mean that the properties preimage resistant and second preimage resistant for MD5 are completely compromised.

SAT Problem. Boolean satisfiability problem (SAT) is the problem of deciding if there is a truth assignment under which a given propositional formula (in conjunctive normal form) evaluates to true. It was shown by Cook [3] that SAT is NP-complete. This was the first problem shown to be NP-complete, and it still holds a central position in the study of computational complexity as the canonical NP-complete problem. The importance of the SAT problem is also grounded in practical applications, since many real-world problems (or their components) in areas such as AI planning, circuit satisfiability and software verification can be efficiently reformulated as instances of SAT. Therefore, good SAT solvers are of great importance and significant research effort has been devoted to finding efficient SAT algorithms.

Due to a general belief that a polynomial time algorithm for SAT is not likely to be found[3] (i.e., it is generally believed that P $\neq$ NP), the only way to evaluate a solver is by its performance on the average, in the worst case, or on a class of SAT instances one is interested in. Also, SAT instances on which the algorithms perform poorly, characterize the weaknesses of these algorithms and can direct further research on improving them.

Experiments suggest that there is a phase transition in SAT problems between satisfiability and unsatisfiability as the ratio of the number of clauses and the number of variables is increased [14]. It is conjectured that, for different types of problem sets, there is a value c_0 of L/N, which is called a *phase transition point* such that:

$$\lim_{N\to\infty} s(N, [cN]) = \begin{cases} 1, \text{for} & c < c_0 \\ 0, \text{for} & c > c_0 \end{cases},$$

where $s(N, L)$ is a *satisfiability function* that maps sets of propositional formulae (of L clauses over N variables) into the segment $[0, 1]$ and corresponds to a percentage of satisfiable formulae. Experimental results also suggest that in all SAT problems there is a typical easy-hard-easy pattern as the ratio L/N is increased, while the most difficult SAT formulae for all decision procedures are those in the crossover region.

zChaff SAT Solver. Majority of the state-of-the-art complete SAT solvers are based on the branch and backtracking algorithm called Davis-Logemann-Loveland algorithm (DLL) [6]. Some of the algorithms also use heuristic local search techniques, but this makes them incomplete (they don't guarantee to find a satisfying assignment if one exists). In addition to DLL, these complete algorithms use a pruning technique called learning. Learning extracts and memorises information from the previously searched space to prune the search in the future. Also, in order to improve the efficiency of the system, techniques as preprocessing, sophisticated branching heuristics, data structures, and random restarts are used (for a survey, see e.g. [20]). There are many SAT packages available, both proprietary and public domain. It is considered that one of the best complete SAT solvers nowadays is the zChaff solver [15]. Besides its smart pruning techniques, zChaff is highly optimised, and achieves remarkably good results in practice. For that reason we chose it as the main SAT solver for our experiments.

[3] Clay Institute for Mathematical Sciences is offering a one million dollar prize for a complete polynomial-time SAT solver or a proof that such an algorithm does not exist (the P vs NP problem).

3 Transforming Cryptanalysis of Hash Functions into SAT Problem

Let *hash* be a hash function generating a hash value of a fixed length N. We assume that *hash* is a hash function with a good distribution of output values. This means that for every (or almost every) N-bit sequence h, there is an N-bit message m having h as the hash value, i.e. $hash(m) = h$). In other words, we assume that the hash function is a good approximation of a permutation on N-bit strings. This holds for hash functions MD4 and MD5, and is important for our investigation. Since the problem of inverting a hash value is highly intractable, in order to scale down the problem hardness we also consider input sequences of length less than N. Let $p_1 p_2 \dots p_M$ denote the bits of an input message (of length M, $M \le N$). The hash function takes this input sequence and transforms it into a sequence of of bits $h_1 h_2 \dots h_N$. For hash functions we are interested in, this transformation is computable and, moreover, expressible in propositional logic, i.e., the resulting hash bits h_i can theoretically be expressed as formulae with $p_1, p_2, \dots, p_M$ as variables. These formulae are very complex as they reflect the inherent complexity of the hash function, but obtaining them effectively is still possible (one method for doing it is described in §4). Let us denote the formula that corresponds to the computation of the bit h_i of the hash value as $H_i(p_1, p_2, \dots, p_M)$.

Preimage Resistance. When analysing preimage resistance of a hash function, the goal is, given a sequence $h_1 h_2 \dots h_N$ (the hash value) and the length of the input message M to determine values $p_1, p_2, \dots, p_M$ that generate this hash value. In other words, we are searching for a valuation v such that $I_v(H_i(p_1, p_2, \dots, p_M)) = h_i \ \ (i = 1, 2, \dots, N)$, where I_v is the interpretation induced by v. Thus, the valuation v must fulfill

$$I_v(H_i(p_1, p_2, \dots, p_M)) = \begin{cases} 1 & \text{if } h_i = 1 \\ 0 & \text{if } h_i = 0 \end{cases} .$$

Further, let $\overline{H}_i$ be defined as

$$\overline{H}_i(p_1, p_2, \dots, p_M) = \begin{cases} H_i(p_1, p_2, \dots, p_M) & \text{if } h_i = 1 \\ \neg(H_i(p_1, p_2, \dots, p_M)) & \text{if } h_i = 0 \end{cases} .$$

Obviously, formula $\overline{H}_i$ is true under valuation v if and only if the hash function *hash* transforms the message corresponding to v into a hash with the i-th bit equal to h_i. The formula $\mathcal{H}$ is defined as follows:

$$\mathcal{H}(p_1, p_2, \dots, p_M) \quad = \quad \bigwedge_{j=1,2,\dots,N} \overline{H}_j(p_1, p_2, \dots, p_M) \ .$$

In order to invert the sequence $h_1, \dots, h_N$, we have to determine a valuation that satisfies the formula $\mathcal{H}(p_1, p_2, \dots, p_M)$. Practically, finding such a valuation is of the same difficulty as to determining whether $\mathcal{H}(p_1, p_2, \dots, p_M)$ is satisfiable.

Hence, we have reduced finding the preimage of a hash function to SAT problem. This reduces the problem of finding the preimage of a hash function to SAT.

Assuming that the hash function is preimage resistant, it is very likely that the formula $\mathcal{H}(p_1, p_2, \ldots, p_M)$ (for large M) is hard to test for satisfiability (otherwise, we would have an effective mechanism for computing the preimage, contradicting the generally accepted assumption of preimage resistance for functions such as MD5). This gives us a method for generating *hard and satisfiable* SAT instances:

1. select a random sequence m of length M;
2. compute the hash value $h_1 h_2 \ldots h_N$ of m;
3. using the above construction, generate the propositional formula $\mathcal{H}$.

Having that valuation induced by m satisfies $\mathcal{H}$ by the construction, it is guaranteed that $\mathcal{H}$ is satisfiable. In addition, it is sound to assume that $\mathcal{H}$ is hard to test for satisfiability. So, this way we can generate *hard and satisfiable* SAT instances for different values of M. Obviously, the bigger M, the harder instance generated.

Second Preimage Resistance. For this property, we assume we are given a sequence $h_1 h_2 \ldots h_N$ (the hash value), the length M of the input message, and also the input bits $p_1, p_2, \ldots, p_M$ that generated this hash value. Our goal is to determine another values $q_1, q_2, \ldots, q_M$ that generate the same hash value. Similarly as above, this reduces to satisfiability of the following formula[4]:

$$\mathcal{H}'(q_1, q_2, \ldots, q_M) \quad = \quad \mathcal{H}(q_1, q_2, \ldots, q_M) \wedge (q_1^{p_1} \vee q_2^{p_2} \vee \ldots \vee q_M^{p_M})$$

where

$$q_i^{p_i} = \begin{cases} \neg q_i & \text{if } p_i = 1 \\ q_i & \text{if } p_i = 0 \end{cases} .$$

The additional clause forces the messages $p_1 p_2 \ldots p_M$ and $q_1 q_2 \ldots q_M$ to differ in at least one bit.

Assuming that the hash function is second preimage resistant, it is very likely that the formula $\mathcal{H}'(q_1, q_2, \ldots, q_M)$ is hard to test for satisfiability for large M. Also, for a good hash function, it is highly unlikely that there is a collision with the length of colliding input messages being less then N. So, it is extremely likely that the formula $\mathcal{H}'(q_1, q_2, \ldots, q_M)$ is unsatisfiable for $M < N$.

This gives us a method for generating *hard and unsatisfiable* SAT instances:

1. select a random sequence m of length M $(M < N)$;
2. compute the hash value $h_1 h_2 \ldots h_N$ of m;
3. using the above construction, generate the propositional formula $\mathcal{H}'$.

This way we can generate *hard and unsatisfiable* SAT instances for different values of M. Obviously, the hardness of generated SAT instances grows with M.

[4] Note that this condition is stronger than the condition given in §2 — namely, the above condition requires that two messages (with the same hash value) have the same length. However, our intention is to use $\mathcal{H}'$ to generate hard unsatisfiable SAT instances and this additional restriction can actually only bring us some good.

Collision Resistance. To check the collision resistance property, one is looking for two different sequences $p_1 p_2 \dots p_M$ and $p'_1 p'_2 \dots p'_M$, with the same hash value. Collision resistance of the hash function can be reduced to satisfiability of the formula

$$\bigwedge_{i=1,\dots,N} (H_i(p_1, p_2, \dots, p_M) \Leftrightarrow H_i(p'_1, p'_2, \dots, p'_M)) \ \wedge \ \neg \bigwedge_{i=1,\dots,M} (p_i \Leftrightarrow p'_i) \ .$$

In this case, the only parameter of the formula is M, the length of the colliding messages we are searching for. The number of variables in the given formula, and hence the complexity of search, doubles with M. This makes these feature too hard and we restricted our investigation only to formulae $\mathcal{H}$ and $\mathcal{H}'$ (described as above).

4 Encoding of Hash Functions into Instances of SAT Problem

It is clear from the previous section how the properties of hash functions can be encoded into SAT instances. In this section we introduce a general framework for such encoding based on existing implementations of hash algorithms. Further, we discuss how to transform the acquired propositional formula into CNF.

4.1 Uniform Encoding on the Basis of Hash Function Implementation

Since a good hash algorithm doesn't depend on the secrecy of the algorithm, all of the popular hash algorithms are available both in a descriptive form and in form of implementations in all popular programming languages. Most of the hash algorithms include thousands of logical operation on input bits. This makes any handcrafting of the propositional formulae we are interested in practically an impossible task. Here we present a framework that allows easy generation of propositional formula of a hash transformation, based entirely on an existing implementation of the algorithm in C/C++[5]. This methodology for encoding cryptographic functions into logical formulae is general and can be applied not only to hash functions, but also other algorithms (e.g., DES). Our approach is considerably simpler, faster and more reliable than one used in [12] where a special hand-crafted program was designed to simulate DES for the same purpose. Also, this approach is independent of a concrete hash algorithm, which makes it readily reusable for further investigations on other cryptographic functions.

The implementation relies on a feature of the C++ language called *operator overloading.* Operator overloading is a specific case of polymorphism, in which

[5] Due to a requirement that hash algorithms must be extremely fast, C/C++ is the most common programming language for implementing hash functions. All available hash functions used in practice are coded in C/C++, so this does not restrict applicability of the method.

operators commonly used in programming such as `+`, `*` or `=`, are treated as polymorphic function, and as such, they can have different behaviours depending on the types of its operators. This feature is usually only a syntactic sugar, and can be emulated by function calls. For example, `x + y * z` can be rewritten as `add(x, multiply(y, z))`. Operator overloading is a common place of criticism when comparing programming languages, since it allows programmers to give to the same operators completely different semantics. This can lead to code that is extremely hard to read, and more important, can lead to errors that are hard to trace. It is considered a good practice to use operator overloading only when necessary and with much care.

We take advantage of offered ambiguous semantics in the following way. Instead of using the algorithm to actually compute the numerical hash value, we change the behaviour of all the arithmetic and logical operators that the algorithm uses, in such a way that each operator produces a propositional formula corresponding to the operation performed. This way one can run the algorithm on general logic variables and produce a formula representing the computation of the algorithm. Afterwards, if needed, one can evaluate the formula obtained by this process, with a specific input message as an argument and get the explicit hash value of the message.

Redefining operators does not (and must not) affect the flow of the algorithm. Since the aim is to record the complete computation of the hash algorithm in one run, this construction can work only if no data-flow dependent conditional structures exist in the code. This restricts the class of the hash functions the approach can handle to the class of linear algorithms (with most, or all, hash functions falling into that category). Some branching and conditional algorithmic structures could also be handled automatically, but it would require more sophisticated interventions in the code together with compiler-like tools that would be able to augment the code appropriately.

Implementation and Overloading of Operators. In the standard implementations of hash functions, 32-bit integers are usually used to represent sequences of 32 bits. To represent a longer bit-array, the array is divided into 32-bit integers, and the computation is entirely performed on these integers. For instance, to represent a sequence of 128 bits of output for MD4 and MD5, four such integers are used. The first step is to create a `Word` data type that would simulate the functionality of 32-bit unsigned integers. The integers in the original implementation will then be replaced by objects of the new `Word` data type. With this data type, each bit of an integer is represented by a propositional formula. These formulae represent a complete logical equivalent of the original computation that produced the value of the integer, one formula per bit. Having this representation, the next step is to define the operators that are used in the hash algorithm for the new data type, so that they consistently represent the propositional counterpart of the expected computation.

Implementation of bitwise logical operators `+` (AND), `|` (OR), `^` (XOR) and `~` (NOT) is straightforward. For example, the overloaded `&` operator takes two `Word` objects and creates a new `Word` object with every bit being an and-formula of the formulae on the corresponding bits of the input objects. These subformulae represent the logic behind the original `&` operator — each output formula is an and-formula of the two corresponding input formulae. For both space and time efficiency, subformulae are not copied into the new formulae, but just linked. See Fig. 1 for our implementation of the `~` operator.

```
Word Word::operator ~ () const {
    Word notWord;  // The not of the word

    /* Compute the not */
    for(int i = 0; i < bitArray.size(); i ++) {
        Formula *f = new FormulaNot(bitArray[i]);
        notWord.setFormulaAt(i, f);
    }

    return notWord; // Return the calculated not
}
```

Fig. 1. Implementation of the `~` operator

For the arithmetic operators the same logic as for logical operators is applied, but the things are a bit more complicated. This complexity arises from the fact that the value of a bit in the result depends on all the previous bits of the operands. See Fig. 2 for our implementation of the `+=` operator.

```
Word& Word::operator += (const Word &w) {
    Formula* c = new FormulaNT;     // The carry bit

    /* Compute the sum, starting from the least significant bit */
    for(int i = bitArray.size() - 1; i >= 0; i --) {
        Formula *andF = new FormulaAnd(bitArray[i], w.bitArray[i]);
        Formula *orF  = new FormulaOr(bitArray[i], w.bitArray[i]);
        Formula *xorF = new FormulaXor(bitArray[i], w.bitArray[i]);

        Formula* sumF = new FormulaXor(xorF, c);  // Sum of the bits and the carry bit
        c = new FormulaOr(andF, new FormulaAnd(c, orF)); // New carry bit of the sum

        setFormulaAt(i, sumF);                    // Set the sum formula at i-th bit
    }

    delete c;     // Delete the last carry
    return *this; // Return the calculated sum
}
```

Fig. 2. Implementation of the `+=` operator

Combining the Implementation with Existing Hash Algorithms. We now have a C++ library that defines the new `Word` data type and implements all the operators that the hash procedure uses. Combining this library with an existing C/C++ implementation of a hash procedure is an easy task. All that is needed is to take the source files and replace the definitions of all integer objects that are used in computation by the newly defined `Word` type. This should suffice to get a running implementation that generates a formula corresponding to the computation of the hash function. One should be careful to avoid replacing the auxiliary objects in the source file, since we are not interested in them, and this would only complicate the computation. Example of such objects are constants, indexes and counters in simple counting loops. Although replacing them will not do any harm to the process, the transformation will be much faster if they retain their original type. See Fig. 3 for example how we modified a part of MD5 implementation to fit our needs. The original MD5 source had `unsigned int` type in places of `Word` type. We tested our implementation and a range of tests (including the original test cases from [16,17]) confirmed its correctness.

```
inline Word MD5Coder::F(const Word &x, const Word &y, const Word &z) {
    return (((x) & (y)) | ((~x) & (z)));
}

inline void MD5Coder::FF(Word& a, const Word& b, const Word& c, const Word& d,
                         const Word& x, unsigned int s, unsigned int ac)
{
    a += F(b, c, d); a += x; a += ac; a <<= s; a += b;
}

void MD5Coder::encodeBlock(int block) {
    ...
    FF (a, b, c, d, message[blockStart + 4], S11, 0xf57c0faf);
    ...
}
```

Fig. 3. Application of the implemented `Word` class on a part of the MD5 algorithm

4.2 Generating Conjunctive Normal Form

In the previous section, we showed how to generate a formula $\mathcal{H}$ corresponding to computation of the hash algorithm. In order to test this formula for satisfiability, one first needs to transform $\mathcal{H}$ into conjunctive normal form (CNF). A propositional formula is said to be in CNF if it is a conjunction of one or more disjunctions of literals. Computing an CNF equivalent of a simple formula is a straightforward but exponential task[6]. There is no unique CNF of a formula, and one can apply several different algorithms to make the CNF transformation (trivial recursive transformation, term-rewriting based, sequent calculus based, etc.). The main problem with these algorithms is that large real world formulae tend to be extremely huge. This, together with exponential complexity of the transformation, makes efficient transformation almost an impossible task.

We tried several standard approaches for computing the CNF equivalent of the formula $\mathcal{H}$, but all of them failed even for relatively small message lengths.

[6] An example of a formula that requires exponential space, and thus also time, is $(p_1 \wedge q_1) \vee (p_2 \wedge q_2) \vee \ldots \vee (p_n \wedge q_n)$.

Either they did not terminate in a reasonable amount of time, or they failed due to high memory requirements. Therefore, we chose a more reasonable approach, which in turn has some other downsides.

Tseitin Definitional Normal Form. If one drops the requirement of equivalence with the original formula, and only keeps SAT-*equivalence*[7], any formula F can be transformed efficiently into a SAT-equivalent CNF. This translation (due to Tseitin [18]) is linear in both the size of the resulting CNF and the complexity of the translation procedure. Since our formulae contain only negation, standard binary logical connectives and, additionally, XOR, the resulting CNF is in $\leq$3CNF form. That is, every clause has at most 3 literals. Generally, any logical formula that contains at most n-ary logical operators can be transformed to $\leq(n+1)$CNF form. This is achieved by introducing a new variable for every logical operation, and then imposing constraints that preserve the semantics of the operation.

We describe the transformation briefly. Let Φ be an arbitrary formula, and let $Sub(\Phi)$ denote the set of all sub-formulae of Φ. For each non-atomic sub-formula $\psi \in Sub(\phi)$, we add a new propositional variable p_ψ. In case ψ is itself atomic, we take $p_\psi = \psi$. Now, consider the formula

$$p_\Phi \wedge \bigwedge_{\substack{\phi \in Sub(\Phi) \\ \phi = \phi_1 \otimes \phi_2}} (p_\phi \Leftrightarrow (p_{\phi_1} \otimes p_{\phi_2})) \wedge \bigwedge_{\substack{\phi \in Sub(\Phi) \\ \phi = \neg\phi_1}} (p_\phi \Leftrightarrow \neg p_{\phi_1}) \quad . \tag{1}$$

It is not hard to see that formula (1) is SAT-equivalent to Φ. The variable p_Φ imposes that Φ is true by propagating the actual evaluation of the formula further up the formula tree using the introduced equivalences.

Transforming formula (1) into CNF is straightforward (see Table 4.2 for transformation rules). Every conjunct in (1) is transformed into CNF with at most 4 clauses, each with at most 3 literals. Thus, the transformation is linear in the size of the formula.

Table 1. Tseitin definitional form transformation rules

Type of formula	Corresponding clauses
$\phi = \neg\phi_1$	$(p_\phi \vee p_{\phi_1}) \wedge (\neg p_\phi \vee \neg p_{\phi_1})$
$\phi = \phi_1 \wedge \phi_2$	$(p_\phi \vee \neg p_{\phi_1} \vee \neg p_{\phi_2}) \wedge (\neg p_\phi \vee p_{\phi_1}) \wedge (\neg p_\phi \vee p_{\phi_2})$
$\phi = \phi_1 \vee \phi_2$	$(\neg p_\phi \vee p_{\phi_1} \vee p_{\phi_2}) \wedge (p_\phi \vee \neg p_{\phi_1}) \wedge (p_\phi \vee \neg p_{\phi_2})$
$\phi = \phi_1 \veebar \phi_2$	$(\neg p_\phi \vee p_{\phi_1} \vee p_{\phi_2}) \wedge (\neg p_\phi \vee \neg p_{\phi_1} \vee \neg p_{\phi_2})$ $\wedge (p_\phi \vee \neg p_{\phi_1} \vee p_{\phi_2}) \wedge (p_\phi \vee p_{\phi_1} \vee \neg p_{\phi_2})$

The main weakness of the definitional CNF transformation is that the number of clauses and variables that are used is quite big. The size of a this CNF form can be reduced significantly by using implications instead of equivalences for

[7] Two formulae F and G are said to be SAT-equivalent when it holds that F is satisfiable iff G is satisfiable (for example p and $p \wedge q$).

subformulae that occur in one polarity only [7]. Applying such optimization here does not yield a considerable decrease, since the vast majority of operations is based on XOR operations, and this makes the subformulae bipolar.

In our case, when analyzing a hash function of fixed length N, since $\mathcal{H}$ is a conjunction of the formulae $\overline{H}_j$, we transform each $\overline{H}_j$ into a CNF, and combine them together into a CNF for $\mathcal{H}$. This yields a CNF with M unit clauses corresponding to each $\overline{H}_j$. These can be eliminated by unit propagation, but we decided to leave this to the SAT solver.

4.3 The HashSAT Formula Generator

We implemented a program that uses the modified implementations of MD4 and MD5 hash algorithms and transforms the formulae $\mathcal{H}$ and $\mathcal{H}'$ to definitional CNF. This program takes various parameters, including a hash function to be used, the length of a message to encode, the number of rounds of the selected hash function to encode (we use this parameter to allow more flexibility on the hardness of SAT instances we generate), etc. The program produces output in the standard DIMACS CNF format[8]. It is a simple textual representation, with one line for each clause. The literals in the clauses are represented as numbers, positive or negative depending on the polarity of the literal. This is a common format for SAT solvers, so the files the program generates can be used as benchmarks for SAT solvers other than zChaff. It is worth noting that this process for generating the formulae is very efficient. For instance, a full MD5 SAT problem for a 128-bit message is generated in under 1.2s with using about 16MB of memory on a Linux 2.60GHz Pentium 4 workstation.

5 Experimental Results

In this section we present our experimental results with SAT formulae generated on the basis of the hash functions MD4 and MD5. We used the methods described in §3 and §4 to generate benchmarks according to the first two properties of hash functions (preimage resistance and second preimage resistance). For each message length M we generated 50 formulae in the way described in §3; bits of starting messages m were generated randomly, each bit taking value 0 or 1 with equal probability.

For the maximal message length the generated problems proved to be too hard to test for satisfiability, so we had to scale down the hardness of the problems. One way is to decrease the message lengths (value M) and hence — decrease the search space. The other approach relies on the inner structure of hash functions: most of hash functions work on equal size block, applying basic transformations grouped in rounds. Both MD4 and MD5 transformations have 4 rounds. By reducing the number of rounds the hash functions become simpler, so the corresponding SAT instances become easier.

[8] For description see `<http://www.satlib.org/Benchmarks/SAT/satformat.ps>`.

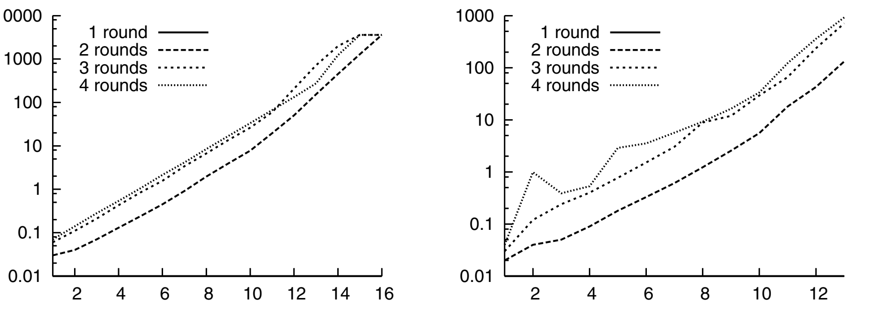

Fig. 4. Preimage resistance (left) and second preimage resistance (right) for MD5. Time scale is logarithmic, CPU time is given is seconds. Separate lines are given for weakened versions with number of rounds from 1 to 3.

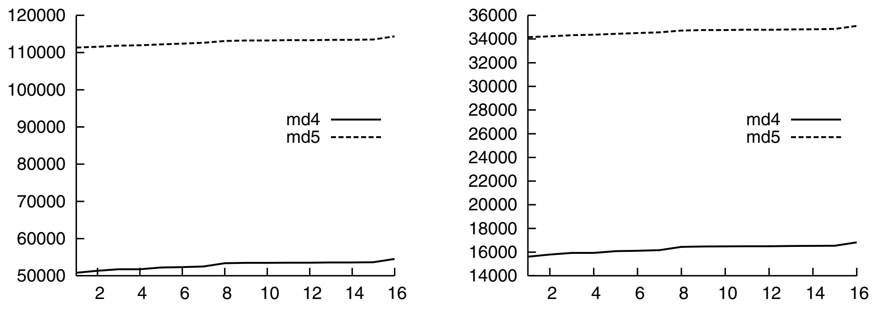

Fig. 5. Growth of clauses (left) and variables (right) in the CNF with size of the input messages for MD4 and MD5

Our results show that, in the worst case, the hash properties, as expected, behave exponentially (against M) when analysed using the described translation to SAT (see Fig. 4). The problems for $M > 16$ needed more time than we set as the time limit (10000s), but we believe that the exponential growth for the CPU time spent, continues for $M > 16$. Note that the restriction to one round gives only trivial problems.

In contrast to the exponential growth for satisfiability testing (in M), the number of clauses grows rather by some small linear factor (see Fig. 5). This means that the we are able to scale the hardness of the formulae arbitrary (for M from 1 to 128), without a significant increase in size of the formula itself.

Figure 6 (left) shows that the function MD4 is more vulnerable to inverting based on the uniform logical analysis. This is expected, as MD5 is generally believed to be of better quality compared to MD4.

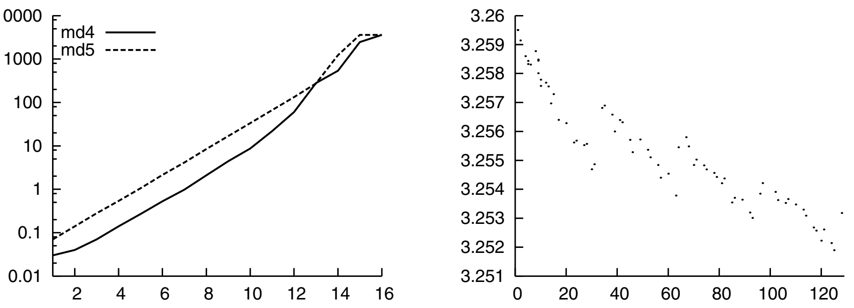

Fig. 6. Comparison of MD5 and MD4 preimage resistance with logarithmic scale for time (left) and L/N ratio with the message length for MD5 (right)

6 Future Work

In the previous sections, we analysed propositional formulae obtained from the input sequences of length M, where $M \leq N$. The value M was used to control the hardness of the problems generated. Alternatively, we could control the problem hardness by the size of the used part of the output: for the preimage resistance feature, we can analyse the following formulae (for the fixed size of the input sequences — the same as the size of the output — N): $\mathcal{H}_i(p_1, p_2, \ldots, p_N) = \bigwedge_{j=1,2,\ldots,i} \overline{H}_j(p_1, p_2, \ldots, p_N)$. Note that the functions $\mathcal{H}_i$ take into account only first i bits of the output. Assuming that the hash function analysed is permutation, for any hash value $h_1 h_2 \ldots h_N$ there is an input of size N that produces it. So, this gives us another method for generating hard and satisfiable SAT instances: (1) select a random bit sequence of length N and use it as a hash value; (2) using the above, construct the propositional formula $\Phi_i = \mathcal{H}_i(p_1, p_2, \ldots, p_N)$. Formulae Φ_i are hard and satisfiable. The bigger i ($i \leq N$), the harder formulae. In a similar manner we could generate unsatisfiable SAT instances. Analysis based on such formulae is the subject of our future research. We will also look at the combinations of these two approaches — controlling problem hardness via both the size of input and the size of output.

One of our motivating ideas for the research presented here was to investigate whether SAT instances generated on the basis of (good) hash functions are among the *hardest instances* (in terms of the phase transition phenomenon in the SAT problem [14]). Unfortunately, we are still unable to answer this question as the generated formulae do not fit the pattern of some class of randomly generated SAT instances, i.e., variables in the clauses are not distributed uniformly. For the generated formulae, the values L/N (number of clauses ratio number of variables) for MD5 are shown in Fig. 6 (right) — these values for $M = 1, \ldots, 128$ are rather stable and range between 3.25 and 3.26 (both with the unit propagation performed or not). Recall that, for some SAT model, the value L/N for the hardest instances is equal to the phase transition point for that SAT model. So, if the variables in our generated formulae were distributed uniformly, and if they indeed the hardest instances (among the formulae with the same distribution on clause length), then the phase transition point for this SAT model would be around 3.25. However, when the unit propagation is performed, in our generated formulae there are around 40% clauses of length 2 and around 60% clauses of length 3 (for $M = 1, \dots, 128$). For such distribution of clause length (and for uniform distribution of variables), [9] approximates the phase transition point at 1.8, and [2] approximates the phase transition point between 2.1 and 2.4, both lower than 3.25. These issues will be subject of our future research — we will try to further investigate the class of generated SAT instances (with non-uniform variable distribution) and whether the instances that correspond to MD5 are indeed the hardest among them. We will try to answer these questions following the ideas from [10].

We are planning to further investigate alternative ways for transforming obtained formulae to CNF (apart from Tseitin's approach) and investigate a possible impact of this on the hardness of generated formulae.

We are also planning to apply the approach presented here to other cryptographic functions (not only the hash functions). For instance, the cryptographic algorithm DES also falls into a category of transformations that can be encoded into propositional formulae by the methodology we propose in this paper.

7 Conclusions

In this paper we presented a novel approach for uniform encoding of hash functions (but also other cryptographic functions) into propositional logic formulae. The approach is general, elegant, and does not require any human expertise on the construction of a specific cryptographic function. The approach is based on the operator overloading feature of the C++ programming language and it uses existing C implementations of cryptographic functions (and needs to alter them only very slightly). By using this approach, we developed a technique for generating *hard and satisfiable* propositional formulae and *hard and unsatisfiable* propositional formulae. Using this technique, one can finely tune the hardness of generated formulae. This can be very important for different applications, including testing (complete or incomplete) SAT solvers. The uniform logical analysis of cryptographic functions can be used for comparison between different functions and can expose weaknesses of some of them (as shown for MD4 in comparison with MD5). We are planning to further develop and apply the technique presented in this paper.

Acknowledgments. We are grateful to anonymous reviewers for useful comments on the first version of this paper.

References

1. D. Achlioptas, C.P. Gomes, H.A. Kautz, and B. Selman. Generating satisfiable problem instances. In *Proceedings of the 17th National Conference on AI and 12th Conference on Innovative Applications of AI*. AAAI Press / The MIT Press, 2000.

2. D. Achlioptas, L. M. Kirousis, E. Kranakis, and D. Krizanc. Rigorous results for random $2 + p$-SAT. Theoretical Computer Science, 265:109–129, 2001.
3. Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71*. ACM Press, 1971.
4. Stephen A. Cook and David G. Mitchell. Finding hard instances of the satisfiability problem: A survey. In *Satisfiability Problem: Theory and Applications*, volume 35 of *DIMACS*. American Mathematical Society, 1997.
5. Ivan Bjerre Damgård. A design principle for hash functions. In *CRYPTO '89*. Springer-Verlag New York, Inc., 1989.
6. Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.
7. Uwe Egly. On different structure-preserving translations to normal form. *Journal of Symbolic Computation*, 22(2):121–142, 1996.
8. Ian Gent. On the stupid algorithm for satisfiability. Technical Report APES-03-1998, Department of Computer Science, University of Strathclyde, 1998.
9. Ian P. Gent and Toby Walsh. The SAT phase transition. In *Proceedings of ECAI-94*, pages 105–109, 1994.
10. I.P. Gent, E. Macintyre., P. Prosser, and T. Walsh. The constraidness of search. In *Proceedings of AAAI-96*, pages 246–252, Menlo Park, AAAI Press/MIT Press., 1996.
11. A. Lenstra, X. Wang, and B. de Weger. Colliding X.509 certificates. Cryptology ePrint Archive, Report 2005/067, 2005. URL: `http://eprint.iacr.org/`.
12. Fabio Massacci and Laura Marraro. Logical cryptanalysis as a SAT problem. *Journal of Automated Reasoning*, 24(1-2):165–203, 2000.
13. Ralph C. Merkle. One way hash functions and DES. In Gilles Brassard, editor, *CRYPTO '89*. Springer-Verlag New York, Inc., 1989.
14. G. David Mitchell, Bart Selman, and J. Hector Levesque. Hard and easy distributions of sat problems. In *AAAI-92*. AAAI Press/The MIT Press, 1992.
15. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *DAC '01*. ACM Press, 2001.
16. Ronald L. Rivest. The MD4 message digest algorithm. RFC 1320, The Internet Engineering Task Force, April 1992.
17. Ronald L. Rivest. The MD5 message digest algorithm. RFC 1321, The Internet Engineering Task Force, April 1992.
18. G. S. Tseitin. On the complexity of derivations in propositional calculus. In *The Automation of Reasoning*. Springer-Verlag, 1983.
19. Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD. Cryptology ePrint Archive, Report 2004/199, 2004. URL: `http://eprint.iacr.org/`.
20. Lintao Zhang and Sharad Malik. The quest for efficient Boolean satisfiability solvers. In *CAV '02*. Springer-Verlag, 2002.