Logical Analysis of Hash Functions

Dejan Jovanović[1] Predrag Janičić[2]

[1]Mathematical institute
Kneza Mihaila 35, 11000 Belgrade, Serbia and Montenegro

[2]Faculty of Mathematics
Studentski trg 16, 11000 Belgrade, Serbia and Montenegro

FroCoS 2005, Vienna, Austria, September 19-21, 2005

Logical Analysis of Hash Functions

Inspired by Massacci, Marraro, Logical cryptanalysis as a SAT problem (2000): ”Encode the low-level properties of state-of-the art cryptographic algorithms as SAT problems and then use efficient automated theorem-proving systems and SAT-solvers for reasoning about them”.

Why?

- New insights into complexity and behaviour of hash functions
- Use cryptographic functions for providing sets of arbitrarily hard benchmarks of industrial relevance for the SAT community

The need for hard SAT problems

Testing Incomplete SAT Algorithms

- Use problem instances that are guaranteed to be satisfiable
- Standard approach: generate problems and use a complete SAT solver to filter out the unsatisfiable instances
- Cannot be used to create problem instances that are beyond the reach of complete search methods

Testing Soundness of Complete SAT Solvers

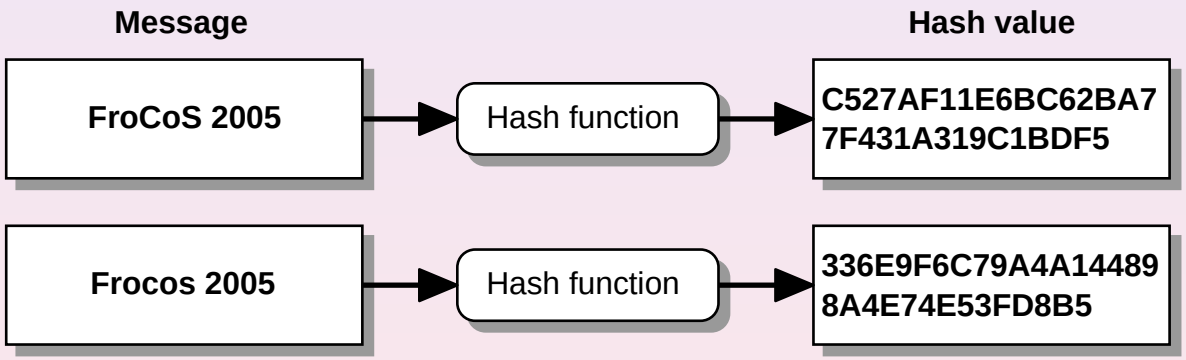
- Use problem instances that are guaranteed to be unsatisfiable

Outline

Outline

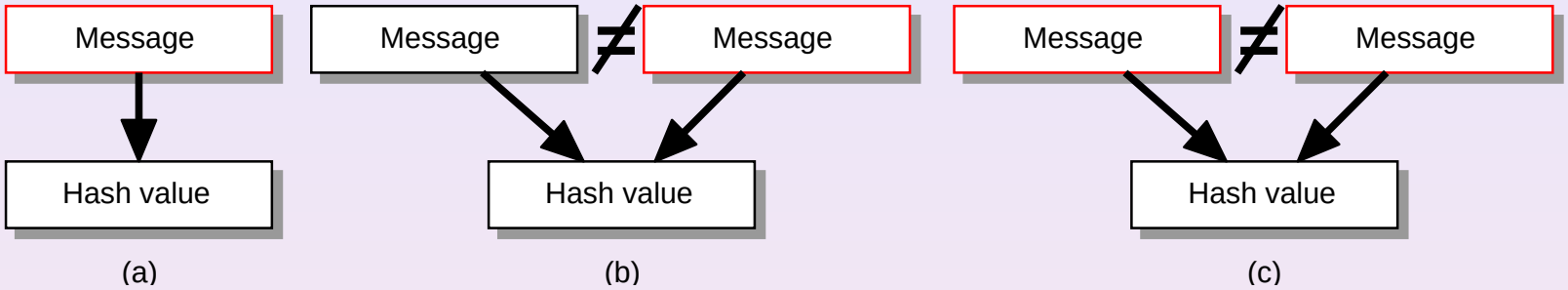
Cryptographic Hash Functions

A cryptographic hash function hash is a transformation that takes an input sequence of bits (**the message**) and returns a fixed-size string, which is called **the hash value** (also the *message digest*, the *digital fingerprint*).



A cryptographic hash function should behave as random as possible and at the same time be deterministic and efficiently computable.

Hash Function Properties



Preimage resistant (a) Given a hash value h, it is computationally infeasible to find a message m such that $hash(m) = h$.

Second preimage resistant (b) Given a message m, it is computationally infeasible to find a message n different from m such that $hash(m) = hash(n)$.

Collision resistant (c) It is computationally infeasible to find two distinct messages m and n such that $hash(m) = hash(n)$.

MD4 and MD5

- MD4 and MD5 are hash algorithms developed by Ronald Rivest in 1990 and 1991
- Both algorithms take a message of arbitrary length and produce a 128-bit hash value
- MD5 is used in many applications, including GPG, Kerberos, TLS / SSL, integrity checks
- Although collisions have been found, preimage and second preimage properties on MD5 are still not compromised

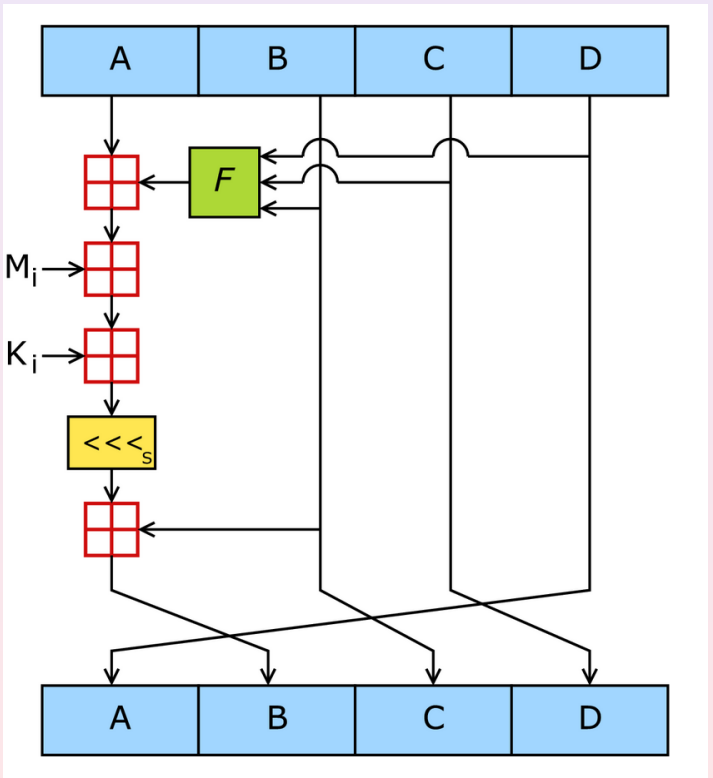
Basic MD5 Operations

MD5 consists of 64 basic operations, grouped in four rounds of 16 operations.

Compression functions

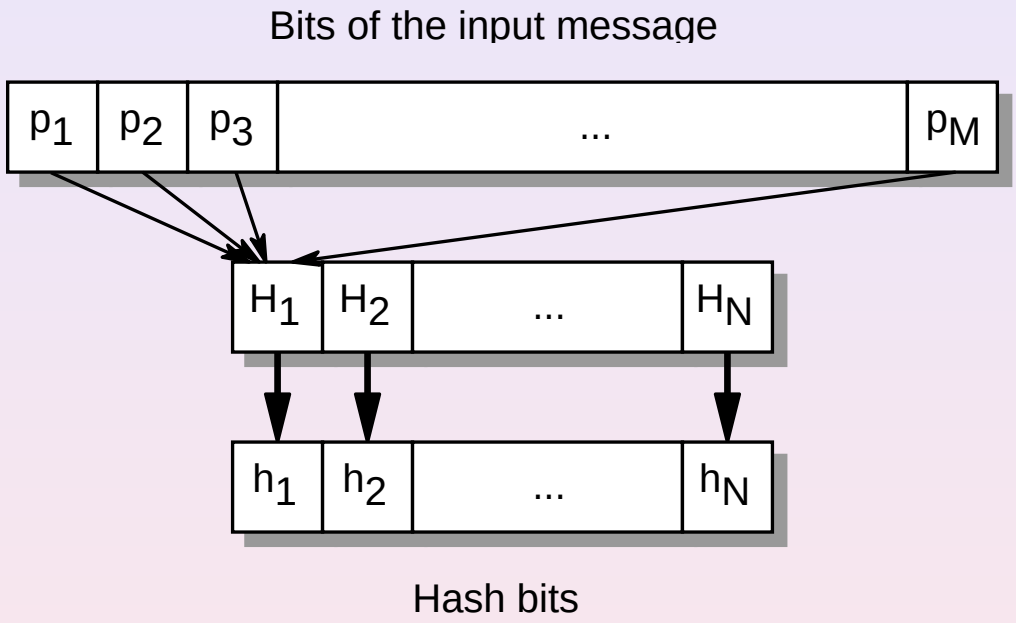
Each round a different function

$$\begin{aligned} \mathbf{F}(x,y,z) &= (x \wedge y) \vee (\neg x \wedge z) \\ \mathbf{G}(x,y,z) &= (x \wedge z) \vee (y \wedge \neg z) \\ \mathbf{H}(x,y,z) &= x \oplus y \oplus z \\ \mathbf{I}(x,y,z) &= y \oplus (x \vee \neg z) \end{aligned}$$



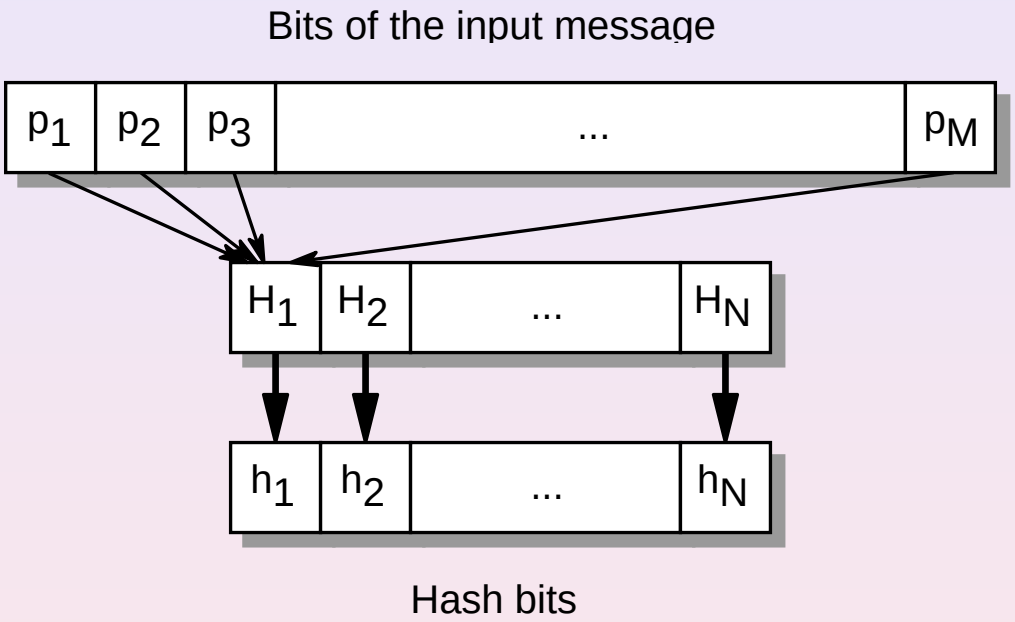
Outline

Transformation to SAT



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

Transformation to SAT



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

Preimage resistance

$$\mathcal{H}(p_1, p_2, \ldots, p_M) = \bigwedge_{i=1,2,\ldots,N} \overline{H}_i(p_1, p_2, \ldots, p_M)$$

- Finding a valuation that satisfies $\mathcal{H}$ is the same as inverting the hash value
- Practically finding such a valuation is of the same difficulty as testing $\mathcal{H}$ for satisfiability
- If hash function is preimage resistant, formula $\mathcal{H}$ should be hard to solve (for large M)

Preimage resistance

Knowing the formulae H_i, we have 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 previous construction, generate the propositional formula $\mathcal{H}$

Second preimage resistance

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

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

- If hash function is second preimage resistant, formula $\mathcal{H}'$ should be hard to solve (for large M)
- It is extremely unlikely to find a collision of a good hash function for small M
- It is extremely likely that $\mathcal{H}'$ is unsatisfiable for small M

Second preimage resistance

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 $\mathcal{H}'$

Outline

Encoding of Hash Functions

How to encode hash functions in propositional logic?

- Hash algorithms include thousands of logical operation on input bits
- Handcrafting of the propositional formulae we are interested in is impossible
- All popular hash algorithms are available as C++ source code

Our approach:

- Use the C++ implementation to generate the formulae automatically
- The approach is independent of a specific hash algorithm

Encoding Based on Hash Function Implementation

Idea

Use the same algorithm to produce the formulae by changing the behaviour of the program using operator overloading.

Operator Overloading

Operator overloading is a specific case of polymorphism, which allows to modify the behaviour of operators commonly used in programming such as $+$, $*$ or $=$, depending on the types of its operators.

Encoding Based on Hash Function Implementation

Let formulae be crated as the program runs

- Provide a new data type to represent integers by formulae
- Each bit in the represented integer is represented by a formula
- Overload all needed arithmetic and logical operators
- Operators take integers represented by formulae and produce integers represented by formulae
- Modify the hash library source code by redefining all integers as the new data type
- Running the hash program will produce the formulae that correspond to hash computation

Example - Source Modification

Source (library)

```
void addSomeNumbers(int x, int y, int& z) {
        z = x + y;
}
```

Source (main program)

```
int x;
int z;
addSomeNumbers(x, 20, z);
```

Example - Source Modification

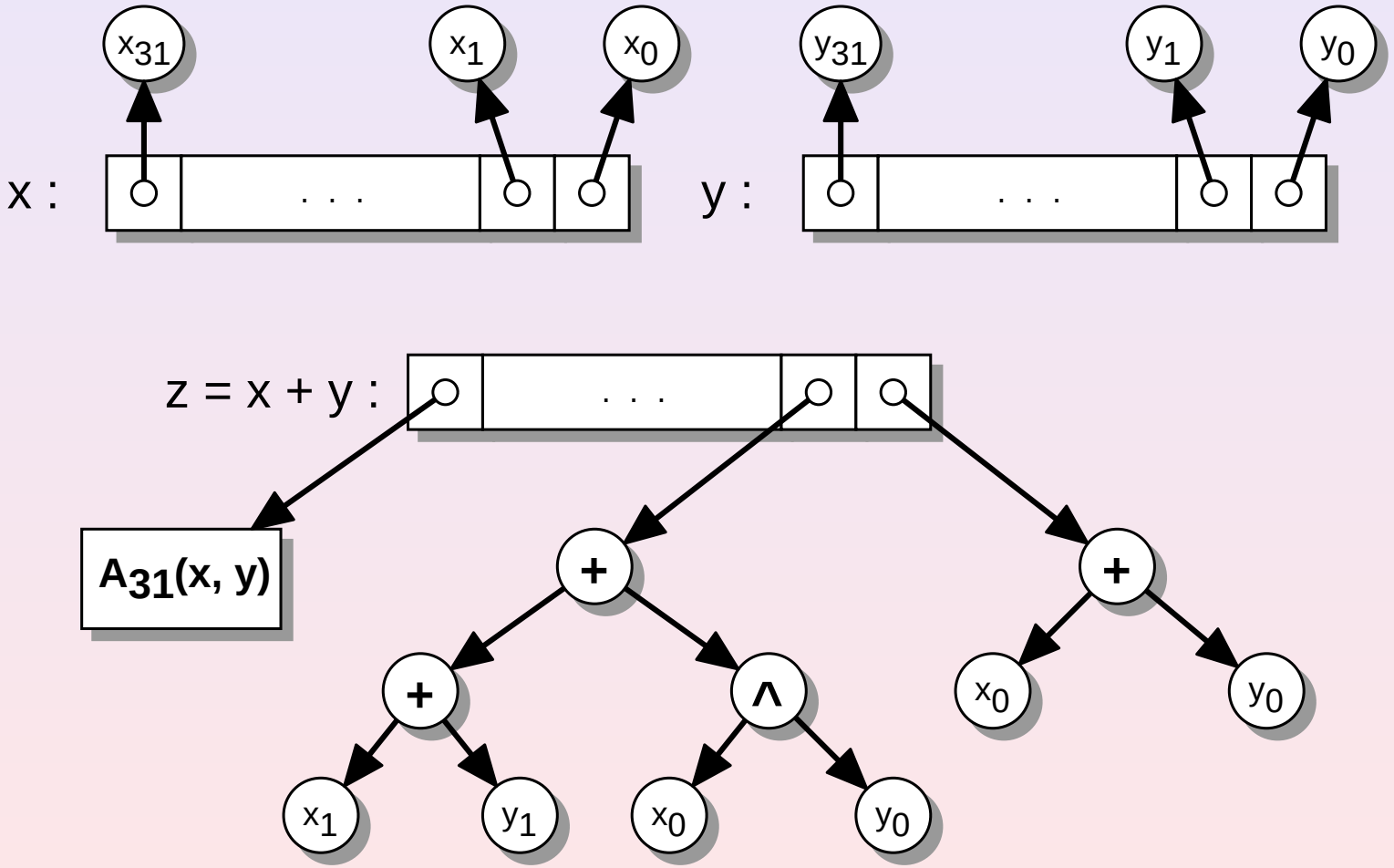
Source (library)

```
void addSomeNumbers(Word x, Word y, Word& z) {
        z = x + y;
}
```

Source (main program)

```
Word x('x', 0, 31);
Word z;
addSomeNumbers(x, 20, z);
```

How does this work?



Our Formula Generator

A program was implemented that uses the modified implementations of MD4 and MD5 hash algorithms and transforms the formulae $\mathcal{H}$ and $\mathcal{H}'$ to definitional CNF.

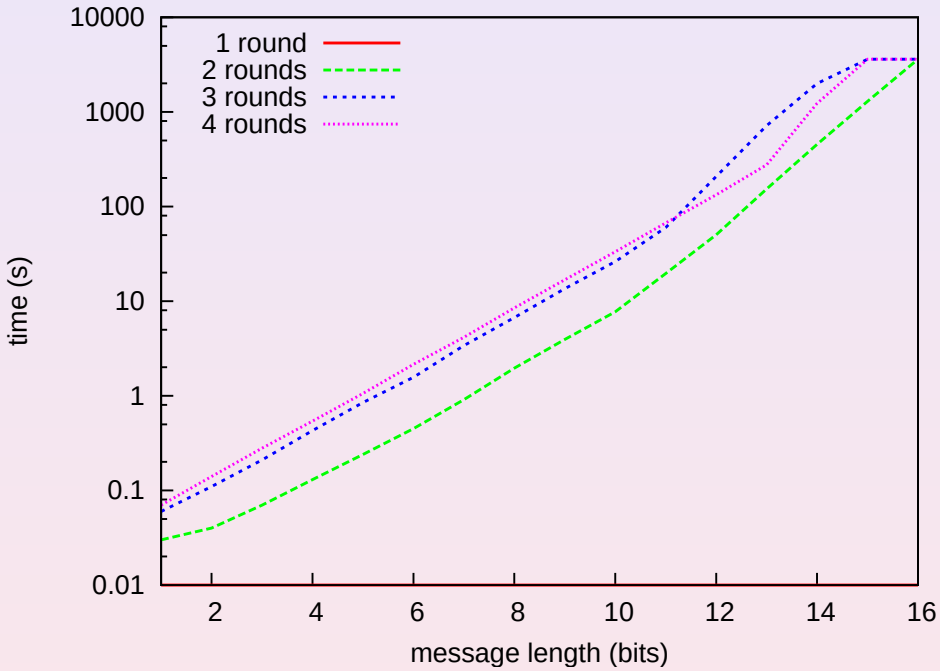
- The implementation was extensively tested and a range of tests (including the original test cases from MD4 and MD5 RFCs) confirmed its correctness
- Program produces SAT problems in DIMACS CNF format
- Generating the formulae is very efficient: full MD5 SAT problem for a 128-bit message is generated in under 1.2s with using about 16MB of memory

Outline

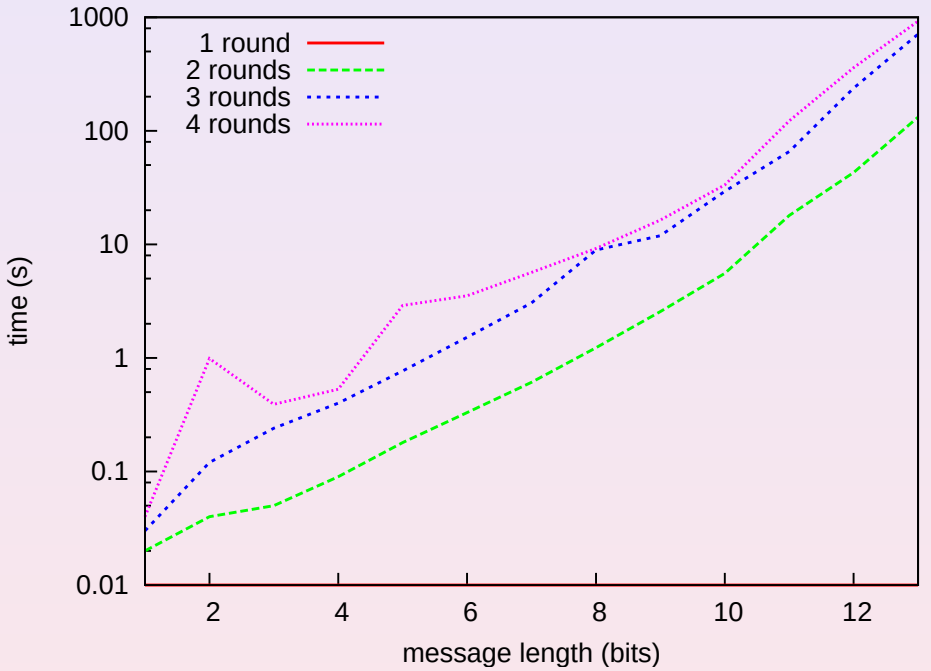
Experimental Results

- zChaff SAT solver was the main solver for our experiments
- Full 128-bit messages were too hard, so we had to scale down the problems using messages up to 16 bits
- Problems were scaled also by number of rounds (1-4 for MD5)
- For each message length M we generated 50 formulae with bits of starting messages m were generated randomly, each bit taking value 0 or 1 with equal probability

MD5 Properties

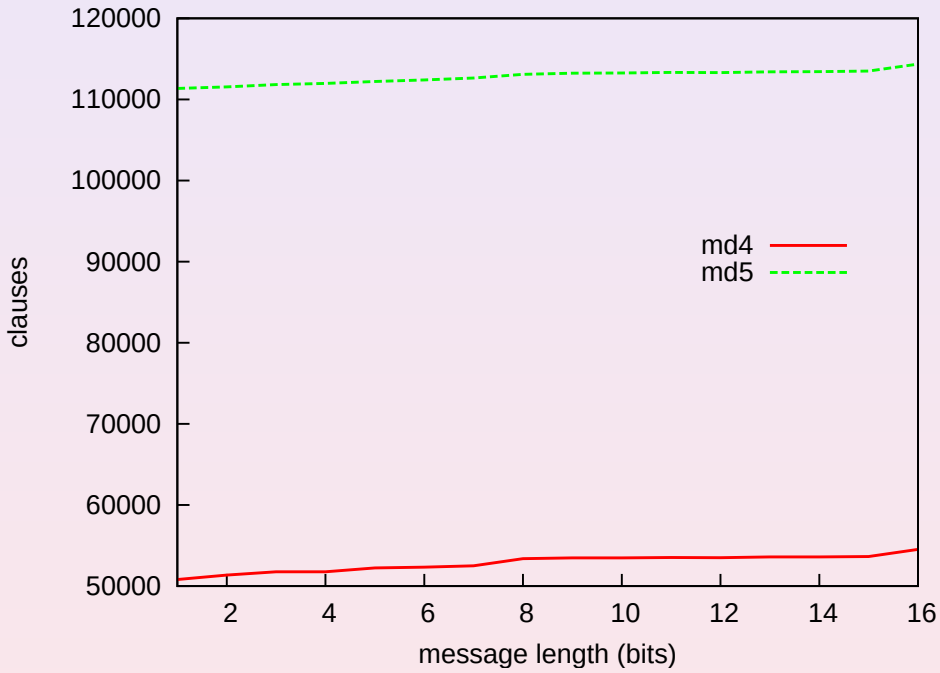


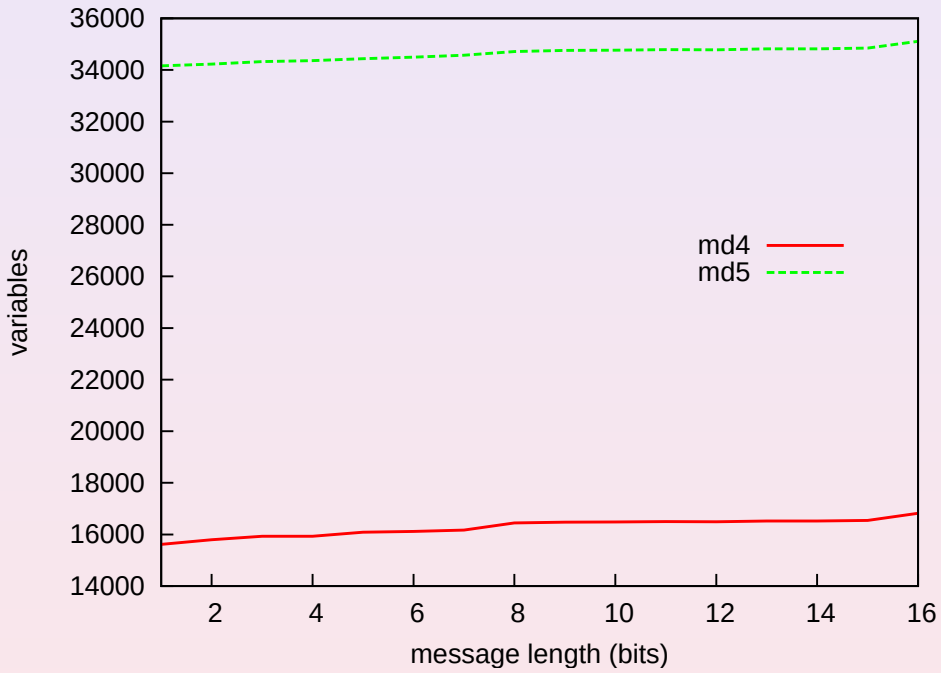
Preimage resistance



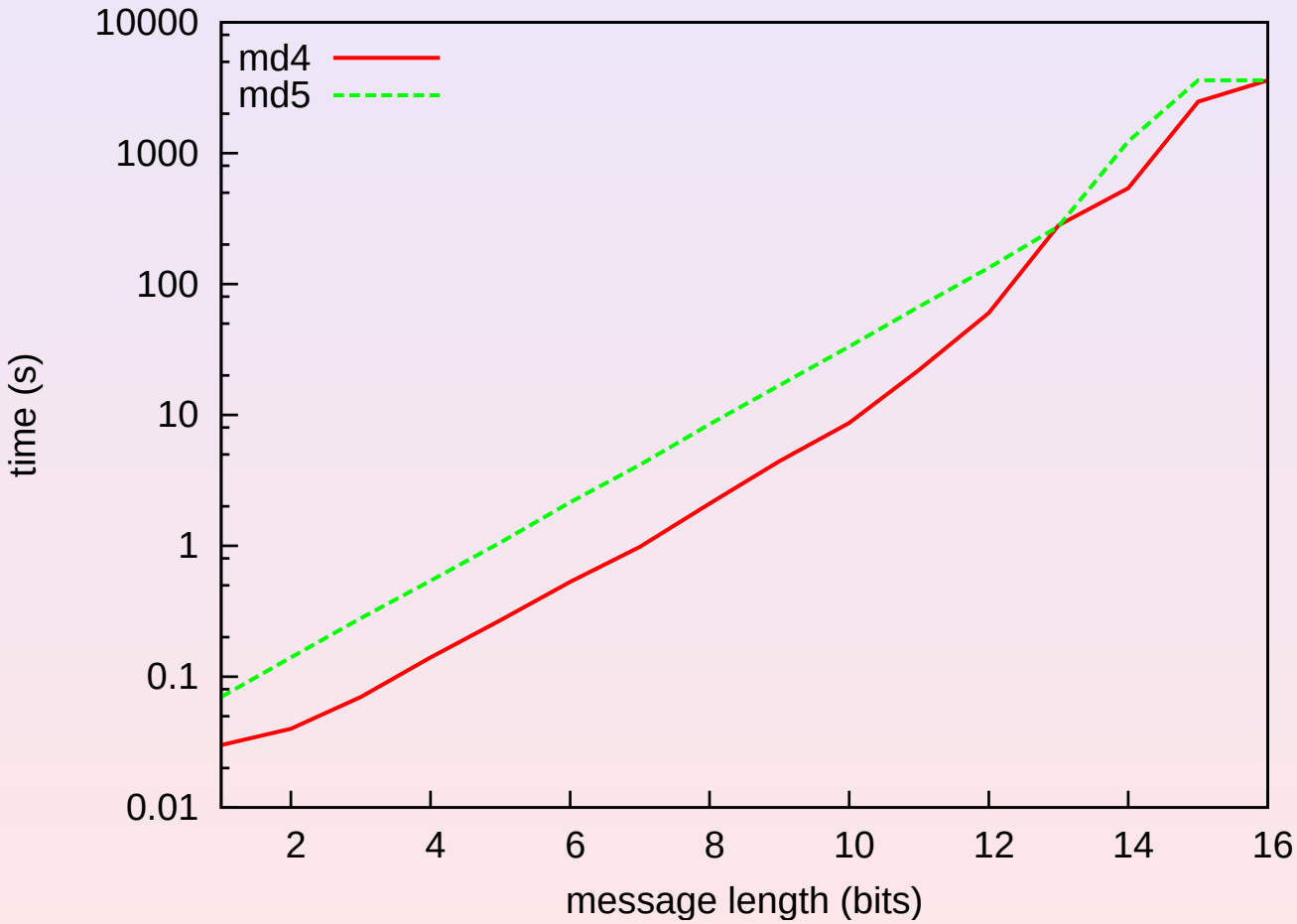
Second preimage resistance

Number of Clauses and Variables





Comparison of MD5 and MD4 Preimage Resistance



Summary

- A novel approach for encoding of hash functions (but also other cryptographic functions) into propositional logic formulae
- Elegant method for generating **hard and satisfiable** and also **hard and unsatisfiable** propositional formulae
- The hardness of formulae can be **finely tuned**
- Applications:
  - Testing of (complete or incomplete) SAT solvers
  - Comparison of different hash functions (as shown for MD4 and MD5)

Future Work

- Control the problem hardness also with output bits
- Experiment with alternative ways for transforming obtained formulae to CNF (apart from Tseitin's approach)
- Investigate whether generated SAT instances are among the hardest SAT instances (in terms of the phase transition phenomenon in the SAT problem)
- Apply the approach presented here to other cryptographic functions (not only hash functions)