How to Fake an RSA Signature by Encoding Modular Root Finding as a SAT Problem [1,2]

Claudia Fiorini [a] Enrico Martinelli [b] Fabio Massacci [b,1]

[a] *Dip. di Informatica e Sistemistica – Univ. di Roma I – ITALY*

[b] *Dip. di Ingegneria dell'Informazione – Univ. di Siena – ITALY*
{*enrico,massacci*}*@dii.unisi.it*

Abstract

Logical cryptanalysis has been introduced by Massacci and Marraro as a general framework for encoding properties of crypto-algorithms into SAT problems, with the aim of generating SAT benchmarks that are controllable and that share the properties of real-world problems and randomly generated problems.

In this paper, spurred by the proposal of Cook and Mitchell to encode the factorization of large integers as a SAT problem, we propose the SAT encoding of another aspect of RSA, namely finding (i.e. faking) an RSA signature for a given message without factoring the modulus.

Given a small public exponent e, a modulus n and a message m, we can generate a SAT formula whose models correspond to the e-th roots of m modulo n, without encoding the factorization of n or other functions that can be used to factor n. Our encoding can be used to either generate solved instances for SAT or both satisfiable and unsatisfiable instances.

We report the experimental results of three solvers, HeerHugo by Groote and Warners, eqsatz by Li, and smodels by Niemela and Simmons, discuss their performances and compare them with standard methods based on factoring.

Key words: Logical Cryptanalysis, Satisfiability, RSA, Modular Cube Roots, Automated Reasoning, Modular Multiplication, Benchmarks

---

[1] Corresponding Author's current address: Fabio Massacci Dip. di Informatica e Telecomunicazioni – Univ. di Trento – via Sommarive 14, 38050 Povo (Trento) - ITALY, massacci@ing.unitn.it.

[2] We would like to thank P. Baumgartner, L. Carlucci Aiello, and the participants to the SAT-2000 workshops for useful comments on this work. C. M. Li provided a compactor for the preprocessing of our formulae before testing. The comments from the anonymous reviewers were most helpful to improve this paper and in particular

1 Introduction

Logical cryptanalysis has been introduced by Massacci and Marraro [28] as a general framework for reasoning about a cryptographic algorithm via a translation into a problem of (propositional) satisfiability on which fast SAT-solvers could be used.

From the viewpoint of automated reasoning, SAT benchmarks based on logical cryptanalysis have a number of advantages:

- their natural formulation requires a fairly rich set of connectives which makes it possible to test formulae beyond CNF;
- they are fairly structured, with abbreviations and definitions, as typically happens for formulae coming from real world applications, such as hardware verification [8];
- problem instances can be randomly generated in almost inexhaustible numbers, by varying either the solution or the instance (while keeping the same solution);
- we can control the solution of the instance without making it too easy to solve, in contrast to standard randomly generated problems for 3-SAT [29,39];
- they are hard to solve and are an excellent test-bed for SAT solvers;
- sometimes they make possible the representation of attacks or properties that are not expressible by traditional cryptography.

In a nutshell, we can use logical cryptanalysis to generate hard, random, structured, solved and controllable instances. Few benchmarks have all these features at once, and few have such a simple intuitive appeal [3].

In [28] Massacci and Marraro applied this approach to the US Data Encryption Standard (DES), a symmetric cipher. Symmetric ciphers seem the natural problem for logical cryptanalysis as the underlying algorithms are mostly based on bit-wise operations. Thus, a translation is a matter of patience, toil, and clever tricks [28].

In constrast, public-key (asymmetric) cryptography is based on number theory and is fairly remote from bit-wise operations. So one may wonder whether

the observation from a reviewer that the straightforward encoding of RSA is not suitable for generating unsatisfiable instances. This work is partly supported by the MURST Project MOSES at the Dipartimento di Informatica e Sistemistica of the University of Roma “La Sapienza”. Fabio Massacci acknowledges the support of a CNR STM fellowship at the University of Koblenz-Landau and at IRIT Toulouse.

[3] The idea of using automated reasoning for breaking a cipher is easier to grasp than the idea of generating SAT problems by instrumenting CAD systems.

it would be possible at all to encode properties of public-key cryptographic algorithms into satisfiability algorithms.

In this paper we concentrate on a particular algorithm: the well known RSA algorithm, proposed by Rivest, Shamir and Adleman [35,43,47]. The idea of using RSA challenges as a test-bed for SAT-solvers was first proposed by Cook and Mitchell [10] who proposed the factorization of large integers as a SAT problem:

> A SAT instance would be an encoding of a boolean multiplier circuit computing the known product M from unknown inputs P and Q. Variables are the bits of P and Q (the inputs of the circuit), together with the outputs of the gates of the circuit. Clauses assert the correct behavior of the gates and assert that the output of the circuit represent the given value of M.[...]
>
> Part of the challenge is to find a suitable multiplier circuit: not too complex and probably not too deep [...].

The first test of SAT solvers on benchmarks of this kind [4] was done by Groote and Warners [19] who use their system HeerHugo. As Cook and Mitchell predicted, SAT-solvers are orders of magnitude slower than ad-hoc factoring methods.

This was to be expected: research on factoring algorithms has few centuries on its side (compared to few decades of SAT research) and it has been recently spurred by the belief that the hardness of factoring is the basis of the security of the RSA crypto-systems. Recently, the RSA challenge 155 (a number with 512 bits) has been just factored by a massive parallel search using an advanced factoring algorithm by te Riele, Cavallar and others [34,47].

Still, the problem of encoding factoring as a SAT problem is simple, compared to the mathematics involved in the RSA crypto-system: the product of two large primes is just the first step, followed by modular reductions, exponentiations, and computations of inverses modulo a congruence.

Cryptographers have spent most of last twenty years in designing either faster factoring algorithms or attacks on RSA which were not dependent on factorization (see the excellent survey by Boneh [5]). Indeed, an intriguing problem for cryptographers is undoubtedly the following: given a message, and a public-key, is it possible to generate a digital signature without knowing the private signature key and without factorization?

The heart of the problem is the computation of the e-th root of a number

[4] The problem was not exactly identical as formulae were added to rule out the trivial factorization of M into 1 and M. Thus, if M is prime the corresponding formula is unsatisfiable.

modulo n, i.e. given three numbers e, n, and m find a number $f < n$ such that $m = f^e \bmod n$. This operation is easy when the factorization of n is known, or when the Euler function [5] $\phi(n)$ is known. If $\phi(n)$ is known or easily computable then we can also easily factor n. So, one is interested in a method for extracting the e-th root modulo n which do not use the factorization of n and that cannot be transformed into an efficient algorithm for factoring n.

This problem is open [6,5]. There are efficient algorithm for computing the e-th root when n is prime but no efficient algorithm has been found when n is composite. Even the most famous "factoring-free" attacks on RSA by Håstad [21] and Coppersmith [11] do not provide general purpose algorithms for computing the e-th root.

1.1 The contribution of this paper

Here, we show how to *encode the computation of e-th root modulo n of a number $m < n$, for a small e into a SAT problem*: if the formula we provide has a model, we can extract from the model a solution of the problem: the bit-wise representation of the root value f such that $m = f^e \bmod n$. If the number we tested for has no e-th root (i.e. m is a e non-residue modulo n in number theory terminology) then the formula is unsatisfiable.

From the viewpoint of the RSA crypto-system this is equivalent to say that we have encoded the problem of faking an RSA signature without recourse to factorization.

To check the effectiveness of SAT techniques on this problem, we have used state-of-the-art SAT provers on our encoding. In particular we have tested our system on

- HeerHugo, by Groote and Warners [19], based on the Stålmark algorithm [20],
- eqsatz, by Li [27], a combination of the traditional Davis-Putnam-Longeman-Loveland (DPLL) procedure [13,14] with equational reasoning for the affine subpart of the problem.
- smodels, by Niemela and Simmons [30,31], an efficient DPLL implementation of the stable model semantics of logic programs that has some features in common with HeerHugo.

Other DPLL implementations have also been tested but the one presented here seemed the most effective. Other approaches to SAT-solving have been ruled out for theoretical reasons: we already know they would have a poor

[5] Number of positive integers smaller than n and relatively prime to n.

performance. For instance, BDD have an exponential blow-up on multiplier circuits [7], and here multiplications are ubiquitous. Local search algorithms such as Walk-SAT are efficient only on problems having solutions with a robust backbone [6] [40], whereas these backbones are extremely fragile: by changing any bit of the solution we do not obtain another solution.

In the experiments on the RSA signature algorithm, we didn't expect to be immediately competitive with advanced algorithms based on number theory and factorization. Still, the result is encouraging: a general purpose search algorithm running on off-the-shelf hardware can crack limited versions of RSA and shows the same behavior of a classical algorithm solving the same problem by factorization (although orders of magnitude slower). Yet, there is a lot of research work that needs to be done since the state-of-the-art version of RSA is still out of reach for SAT-based systems [7]

Anyhow, we would not like to stress that "SAT-based attack" point beyond reasonable. Indeed, even if trying to "beat a number theorist at his own game" is tempting, our main interested is SAT research and not number theory and we should look to this problem from the standpoint of SAT-research.

In this respect, we have already stressed the motivations behind our advocacy of logical cryptanalysis as a SAT benchmark: it provides a set of challenging problems of industrial relevance as asked for in [38], a hierarchical and regular structure with abbreviations and definitions, and large affine subproblems (i.e. formulae with exclusive or), gives the possibility of generating as many random instances as one wants of both satisfiable and unsatisfiable nature [8].

Benchmarks from logical cryptanalysis stretch system performance. Systems that performs well on such problems are indeed likely to perform well on many other real-word problems (as it is indeed the case for eqsatz [27] on the DIMACS parity bit challenge).

[6] Loosely speaking, the backbone of a satisfiable formula is the set of variables having the same truth-value in all solutions (satisfying assignments) of the formula.
[7] To be precise it is also out of reach for number theoretic algorithms as all known factoring algorithms are sub-exponential [9,47] and require massive parallelization to be effective.
[8] Strictly speaking, encoding the falsification of RSA signatures can only generate satisfiable instances, but we show that with minor modifications also unsatisfiable instances can be generated.

1.2 Plan of the paper

In the rest of the paper we briefly introduce RSA (§2). then we present the basic ideas behind logical cryptanalysis of RSA by encoding cubic root extraction as a SAT problem (§3) and give the detail of the encoding (§4). We explain how to generate satisfiable and unsatisfiable instances (§5) and report of our experimental analysis (§6). Brief conclusions (§7) end the paper.

2 A Primer on RSA

RSA is a public-key crypto-system developed in 1977 by Ron Rivest, Adi Shamir and Leonard Adleman [35]. It is a widely used algorithm for providing privacy and ensuring authenticity of digital data. The RSA system uses modular arithmetic to transform a message (represented as a number or a sequence of numbers) into unreadable ciphertext.

Here we give the essential mathematical features of the algorithm and refer to the PKCS#1 standard by RSA Security [36] or to general books on computer security [37,43,47] for the technical complications arising in practical implementations such as padding messages with random strings, using hash functions, etc.

Definition 1 *Let* $n = p \cdot q$ *be the product of two large primes. Let* e *be an integer co-prime with* $\phi(n) = (p-1)(q-1)$*, the Euler function of* n*. Let* d *be the integer solution of the equation* $ed = 1 \bmod \phi(n)$*. We call* n *the* modulus, e *the* public exponent *and* d *the* private exponent*. The pair* $\langle e, n \rangle$ *is called* public key *and the pair* $\langle d, n \rangle$ *is the corresponding* private key*.*

The public key is widely distributed whereas the private key must be kept secret.

Definition 2 (Message Signature and Verification) *Let* $\langle e, n \rangle$ *be a public key and* $m < n$ *be a message. To* sign [9] m*, the agent holding the corresponding private key pair* $\langle d, n \rangle$ *computes the integer* f *such that* $f = m^d \bmod n$*. To* verify *that* f *is the signature of a message* m *one computes* $m' = f^e \bmod n$ *where* $\langle e, n \rangle$ *is the public key, and accepts* f *as valid only if* $m' = m$*.*

The intuition is that we can be sure that the signature is authentic because only the holder of the private key could have generated it, though everybody can verify it by using the public key.

[9] In the sequel we use the letter f for the italian word "firma" for signature.

In practical applications, a typical size for n is 1024 bits, so each factor is about 512 bits long. As for the sizes of e and d, it is common to choose a small public exponent for the public key, to make verification faster than signing. For instance, several security standards [24,3,36] recommend either 3 or 65537 (corresponding to $2^{16} + 1$), without significantly degrading the RSA security level.

For a "total break" of the RSA crypto-system an attacker needs an algorithm for recovering the private exponent d from a public key $\langle e, n \rangle$: this would enable him to forge signatures at wish. For a "local deduction", it is sufficient to recover the signature f of a given message m using only the knowledge of m and the public key $\langle e, n \rangle$.

A total break can be obtained only by finding an efficient algorithm for factoring the modulus n into its two prime factors p and q: from p, q and e it's easy to get d, the private exponent using Euclid's greatest common divisor algorithm. The converse is also true: from d one can efficiently recover the factorization of n [5,9,47].

We cannot discuss here the various flavours of factoring algorithms and refer to [25,9,47] for details. We just note that the best general-purpose factoring algorithm today is the probabilistic *Number Field Sieve*, which runs in expected time $O(e^{1.9223(\ln n)^{1/3}(\ln \ln n)^{2/3}})$. Older methods are usually faster on "small" numbers: for instance, Pollard's ρ method is better for numbers having small factors (say up to 10 decimal digits) and the Elliptic curve method works well for finding factors up to 30-40 digits [26].

The most important observation about factoring is that all known algorithms require at least a sub-exponential amount of time in the number of bits of the modulus and the most effective ones also sub-exponential space [9,47,41]. The last RSA Challenge that was factored is a 512-bit modulus [34]. The total amount of computer time spent using the Number Field Sieve factoring algorithm was estimated to be the equivalent of 8000 MIPS-years. So, the state of the art in factoring is still far from posing a threat when RSA is used properly [41].

Thus, a number of researchers have worked on methods that try to decrypt messages or obtain signatures of messages without factoring the RSA modulus n. Some of these attacks, most notably those due to Håstad [21] and Coppersmith [11], make it possible to fake signature or to decrypt particular messages without factoring the modulus under certain circumstances.

The Common Modulus Attack on RSA is an example of a local deduction: a possible RSA implementation gives everyone the same n, but different values for the exponents e and d. However, if the same message is ever encrypted with two different exponents with same modulus, then the plaintext can be recov-

ered without either of the decryption exponents. This and similar attacks can be thwarted by suitably padding messages with independent random values.

Yet, there is no algorithm for performing local deductions in the general case: given an arbitrary (small) public exponent e, a modulus n, and a message m, compute a signature f such that f would pass the verification test $m = f^e \bmod n$, without the preliminary factorization of n.

If we "invert" the $m = f^e \bmod n$ equation, into $\sqrt[e]{m} \bmod n = f$, we can see that we just need an algorithm to extract the e-th root modulo n of an integer $m < n$. If n is prime, many efficient algorithms are known, either using direct methods or based on the index calculus [9,47, Sec. 1.6]. For instance, if $e = 2$ and n is prime we can compute square roots in time $O(\log^4 n)$. However, when n is composite, and even if e is a small number such as 3, there is no general method for finding e-th roots which does not use the factorization of n (or something that can be used for efficiently factoring n). See [5,6] for a more comprehensive discussion.

3 Logical Cryptanalysis of RSA

The main intuition behind logical cryptanalysis as introduced in [28] is that we should represent a cryptographic transformation $C = E_K(P)$, where P is the plaintext, C is the ciphertext, and K is the key, with a suitable logical formula.

If we choose propositional logic, then we must encode each bit sequence P, C, K as a sequence of *propositional variables* $\mathbf{P}$, $\mathbf{C}$, $\mathbf{K}$, in which every variable is true when the corresponding bit is 1 and false when it is 0. Then the properties of the transformation are encoded with a logical formula $\mathcal{E}(\mathbf{P}, \mathbf{C}, \mathbf{K})$ which is true if and only if the cryptographic transformation holds for the corresponding bit sequences.

For a symmetric cipher such as DES, the choice of the cryptographic transformation is almost obvious (the encryption or decryption algorithm) and the difficult part is just the translation.

For RSA, the situation is not so simple: we have three known values e, n and m, and a number of equations to choose from.

If we choose $n = p{\cdot}q$, we can represent factoring as a SAT problem as suggested by Cook and Mitchell [10]. The hardness of this SAT problem have been already investigated by Groote and Warner [19].

Since a "total break" of the algorithm is unlikely, we might prefer to encode

the computation, via a SAT encoding, of the e-th root modulo n of m. In this case we have two options:

$$f = m^d \bmod n \text{ holds} \quad \Leftrightarrow \quad \mathcal{R}SA(\mathbf{m}, \mathbf{d}, \mathbf{f}, \mathbf{n}) \text{ is true} \tag{1}$$
$$m = f^e \bmod n \text{ holds} \quad \Leftrightarrow \quad \mathcal{R}SA(\mathbf{f}, \mathbf{e}, \mathbf{m}, \mathbf{n}) \text{ is true} \tag{2}$$

The first choice seems to be preferable as a model of the formula yields a value for the private key. Unfortunately, it has too many unknowns and therefore too many solutions which would not pass the verification test $m = f^e \bmod n$. For instance, if we set $\langle 3, 55\rangle$ as the public key and 9 as the message, we could find $16 = 9^4 \bmod 55$ as a solution of the equation $f = m^d \bmod n$ but unfortunately $9 \neq 16^3 \bmod 55 = 26$.

Thus, the solution seems just picking up a combinatorial circuit that takes e, f and n as inputs and has $m = f^e \bmod m$ as output. Then, we could just "update" Cook and Mitchell's idea: "variables are the bits of e, f, and n (the inputs of the circuit), together with the outputs of the gates of the circuit. Clauses assert the correct behavior of the gates and assert that the outputs of the circuit represent the given value of m."

This simple idea turns out to be unfeasible: there is no combinatorial circuit for modular exponentiation. The algorithms used in practice reduce it to a sequence of modular multiplications based on the principle of "square-and-multiply" [9,47]. Loosely speaking we may represent this procedure as follows:

$$m_0 = 1$$
$$m_{i+1} = (m_i^2 + e_i \cdot f) \bmod n$$

where e_i is 0 (resp. 1) if the corresponding i-th bit of the binary representation of e has the value 0 (resp. 1). The desired value m is obtained at $m_{\lfloor \log e \rfloor + 1}$.

If we assume that e can be arbitrary, our encoding into satisfiability must take into account the largest possible value of e (that is $\log e \sim \log n$). Then, we must encode $\log n$ modular multiplications most of which will turn out to be just useless as they would not be activated.

If we are just worried about correctness, this problem is immaterial. The fragment of the formula corresponding to the inactivated modular multiplication steps will not change the solutions of the problem: any model of the formula would still yield an e-th root of m modulo n. When $\mathbf{v}_e$ is replaced for $\mathbf{e}$ in the formula, unit propagation will set the appropriate values of the inputs of each m_i.

The major problem is that, from the viewpoint of the SAT solver this would be a disaster. The solver has no way to know that we do not care of the values of

variables corresponding to inactivated modular multiplication steps. Syntactic analysis would not help it, since one of the operands of the multiplication is the signature f. The solver will have to search in that subspace too and we have no guarantee that the search heuristics will recognize that this search is actually pointless.

Moreover, the size of the problem would become huge even for small n. If we assume that the square-and-multiply step can be encoded using the best possible multipliers and only $O(\log n \log\log n)$ gates, the encoding of the RSA signature algorithm for a modulus of 100 bits would require over 100.000 formulae. If we use standard parallel multipliers we would need over 1.000.000 formulae.

So a smarter encoding is needed. At first, it should be possible to introduce more variables in $\mathcal{RSA}(\mathbf{f}, \mathbf{e}, \mathbf{m}, \mathbf{n})$ besides $\mathbf{m}$, $\mathbf{f}$, $\mathbf{e}$, and $\mathbf{n}$, and make use of abbreviations and definitions. Still, if the encoding is well designed, then $\mathbf{m}$, $\mathbf{f}$, $\mathbf{e}$, and $\mathbf{n}$ should be the only *control variables* of the problem, i.e. fixing their values should determine the values of all other variables. Thus, if we replace the variables $\mathbf{m}$, $\mathbf{e}$, and $\mathbf{n}$ with their respective 0/1 values $\mathbf{v}_m$,$\mathbf{v}_e$, and $\mathbf{v}_n$ the control variables in $\mathcal{RSA}(\mathbf{f}, \mathbf{v}_e, \mathbf{v}_m, \mathbf{v}_n)$ should be only $\mathbf{f}$.

Another desirable property of the encoding is that it should be possible to use $\mathcal{RSA}(\mathbf{v}_f, \mathbf{v}_e, \mathbf{m}, \mathbf{v}_n)$ and unit propagation to directly compute $\mathbf{v}_m$.

Our solution is to use the same trick used for the encoding of DES with a variable number of rounds [28]: given e, run the algorithm at the meta-level and encode only the modular multiplications which actually take place.

Rather than using the correspondence set in (2) we must use the following one

$$m = f^e \bmod n \text{ holds} \quad \Leftrightarrow \quad \mathcal{RSA}_e(\mathbf{f}, \mathbf{m}, \mathbf{n}) \text{ is true} \tag{3}$$

where the value of e is a parameter of the encoding.

4 Encoding Modular Exponentiation into SAT

So far, we have reduced ourselves to the problem of representing in logic the modular congruence in (2), where the value of e is handled differently from the values of m, f and n in the encoding.

As we have already noted, the public exponent e does not need to be a very large number, and security standards [24,3,36] recommend a value such as

3 or 65537 (corresponding to $2^{16}+1$), thus limiting the number of modular multiplications to respectively 2 or 17.

For sake of simplicity, and since the particular problem is as hard as the general one [5,9,47], in this work we choose the value $e = 3$ and consequently equation (2) can be rewritten in the form

$$m = ((f \cdot f) \bmod n \cdot f) \bmod n \tag{4}$$

According to (4), the exponentiation can be plainly carried out by iterating a modular multiplication twice.

At first, emphasis should be on efficient implementations of modular multipliers. If we look at modular multipliers, many designs have been proposed in the literature, ranging from look-up table based structures for small moduli [42,23,32,15], to devices restricted to specific moduli [33,44,22], to architectures suitable for medium and large moduli and using only arithmetic and logic components [16,1,45,2,22].

Also in this case, just picking up a multiplier would not do the job:

- f and the factors p and q of the modulus n are *unknown* in our setting and this makes solutions in which they are hardwired in the implementation impossible to use;
- we must use a *purely combinatorial multiplier* since sequential operations [10] are not readily representable in propositional logic;
- the implementation must reduce the use of number theoretic features as much as possible (relatively simple number theoretic operations like the Euclid greatest common divisor algorithm are difficult to encode into propositional logics).

The constraints rule out many of the recent algorithms for modular multiplication published in the literature.

Thus, we singled out the multiplication structure described in [1], since it features a simple purely combinatorial formulation.

The basic intuition behind the method is the following: given two integers x and y represented in the range $[0, n)$ by $b = \log n$ bits, the multiplication of x times y modulo n yields a non negative integer π such that

$$\pi = (x \cdot y) \bmod n = x \cdot y - k \cdot n$$

[10] A sequential implementation of modular exponentiation repeatedly applies the combinatorial algorithm until some condition is reached.

where $k = \lfloor (x \cdot y)/n \rfloor$. Note that $x \cdot y$ requires $2b$ bits for representation, while π is still represented on b bits.

We can see that the computation can be reduced to compute the integer k. Once we get the value of k right, the rest are just the standard operations of addition, subtraction and multiplication between integers.

However, division is a complex operation and it is simpler to compute an approximate value of k and then subtract the error. So, we split the computation of k into the product of $(x \cdot y) \cdot (1/n)$ and approximate the computation of $1/n$ by a fraction t having as many digits as required to evaluate k by a number k_{ap} differing from the true value by at most 1. Namely, if t is the number obtained limiting $1/n$ at the first r fractional bits, the following inequalities hold:

$$t \leq 1/n < t + 2^{-r}$$

and multiplying both sides by $x \cdot y$

$$x \cdot y \cdot t \leq (x \cdot y)/n < x \cdot y \cdot t + x \cdot y \cdot 2^{-r}$$

From the latter inequalities it is immediate to see that replacing $\lfloor (x \cdot y)/n \rfloor$ by $(x \cdot y) \cdot t$ the maximum error is bounded by $E = x \cdot y \cdot 2^{-r}$ and imposing the constraint $E < 1$, it must be $r \geq 2b$. Indeed, under this assumption, we have

$$E < 2^{2b} \cdot 2^{-r} \leq 1.$$

Denoting the value $\lfloor x \cdot y \cdot t \rfloor$ by k_{ap}, we can write:

$$\begin{aligned} k &= \lfloor \tfrac{x \cdot y}{n} \rfloor = \lfloor x \cdot y \cdot t + E \rfloor = \lfloor k_{ap} + Fract(x \cdot y \cdot t) + E \rfloor = \\ &= k_{ap} + \lfloor Fract(x \cdot y \cdot t) + E \rfloor = k_{ap} + E' \end{aligned}$$

Since $0 \leq Fract(x \cdot y \cdot t) + E < 2$, E' must be 0 or 1.

Finally, the modular product π is expressed by the relation

$$\pi = x \cdot y - k_{ap} \cdot n - E' \cdot n$$

To obtain π, the expression $x \cdot y - k_{ap} \cdot n$ must be computed and tested against n: if it is less than n, it is correct ($E' = 0$); otherwise n must be further subtracted. Note that always $0 < x \cdot y - k_{ap} \cdot n < 2n$, that is this value is correctly represented by means of $b + 1$ bits and thus only the $b + 1$ less

significant bits of $x \cdot y$ and of $x \cdot y - k_{ap} \cdot n$ are necessary for the computation, reducing the formula complexity.

The structure that implements the described algorithm is shown in Fig. 1: it mainly consists of three binary multipliers and two binary adders. Multiplier MUL1 produces $x \cdot y$, that in turn is multiplied by constant t through a $2b \times 2b$ multiplier MUL2. The $4b$-bits MUL2 output contains the representation of k_{ap} in the interval from bit position $2b - 1$ to bit position $3b$. This representation is multiplied by constant $-n$ by multiplier MUL3 to produce $x \cdot y - k_{ap} \cdot n$. Finally, values $x \cdot y - k_{ap} \cdot n$ and $x \cdot y - k_{ap} \cdot n - n$ are yielded by adders ADD1 and ADD2, and the final result π is chosen, depending on the sign of the latter value.

The algorithmic structure of Fig. 1 can be easily expressed in the notation of proposition logic, joining sub-expressions drawn for single components.

As for adders and multipliers, the former are modeled by ripple-carry adders. At first, this choice may appear rather inefficient in respect of faster solutions, like carry look-ahead adders or carry-save adders (CLA or CSA), but our preliminary experiments showed that the superior circuit performance does not guarantee a similar efficiency of SAT solvers on the encoding. This phenomenon is also frequent in hardware verification [8]: simpler and unoptimized circuits are easier to analyze than optimized ones because the latter use complex boolean functions. A b-bit ripple carry adder is described in Fig 2 joining the bit-wise equations of a full adder cell, where A_i, B_i are the i-th bits of operands and C_i, C_{i+1} are the i-th and the next carry bits.

Array multiplying structures were chosen for multiplication. The multiplication of two b-bits numbers can be easily implemented by an array with b rows and $2b - 1$ columns of full/half adders, as shown in Fig. 3 in the case $b = 3$. Generalizing the structure of Fig. 3, it is easy to derive the set of boolean expressions for an $n \times n$ array multiplier shown in Fig. 4.

As for multipliers, one may again argue that one could have used Wallace multipliers or the recursive construction due to Karatsuba. We ruled them out for the same reasons that led us to prefer ripple carry adders to carry-lookahead adders.

Finally, the expressions in (2) and (4) are combined to produce the logical description of the operation $\Pi(\pi, \mathbf{x}, \mathbf{y}, \mathbf{n})$ representing the modular product $\pi = (x \cdot y) \bmod n$. Iterating the process twice, the final result is

$$m = f^3 \bmod n \text{ holds } \Leftrightarrow \quad \mathcal{R}SA_3(\mathbf{f}, \mathbf{m}, \mathbf{n}) \text{ is true} \quad \Leftrightarrow$$
$$\Leftrightarrow \Pi(\mathbf{m}', \mathbf{f}, \mathbf{f}, \mathbf{n}) \wedge \Pi(\mathbf{m}, \mathbf{f}, \mathbf{m}', \mathbf{n}) \text{ is true}$$

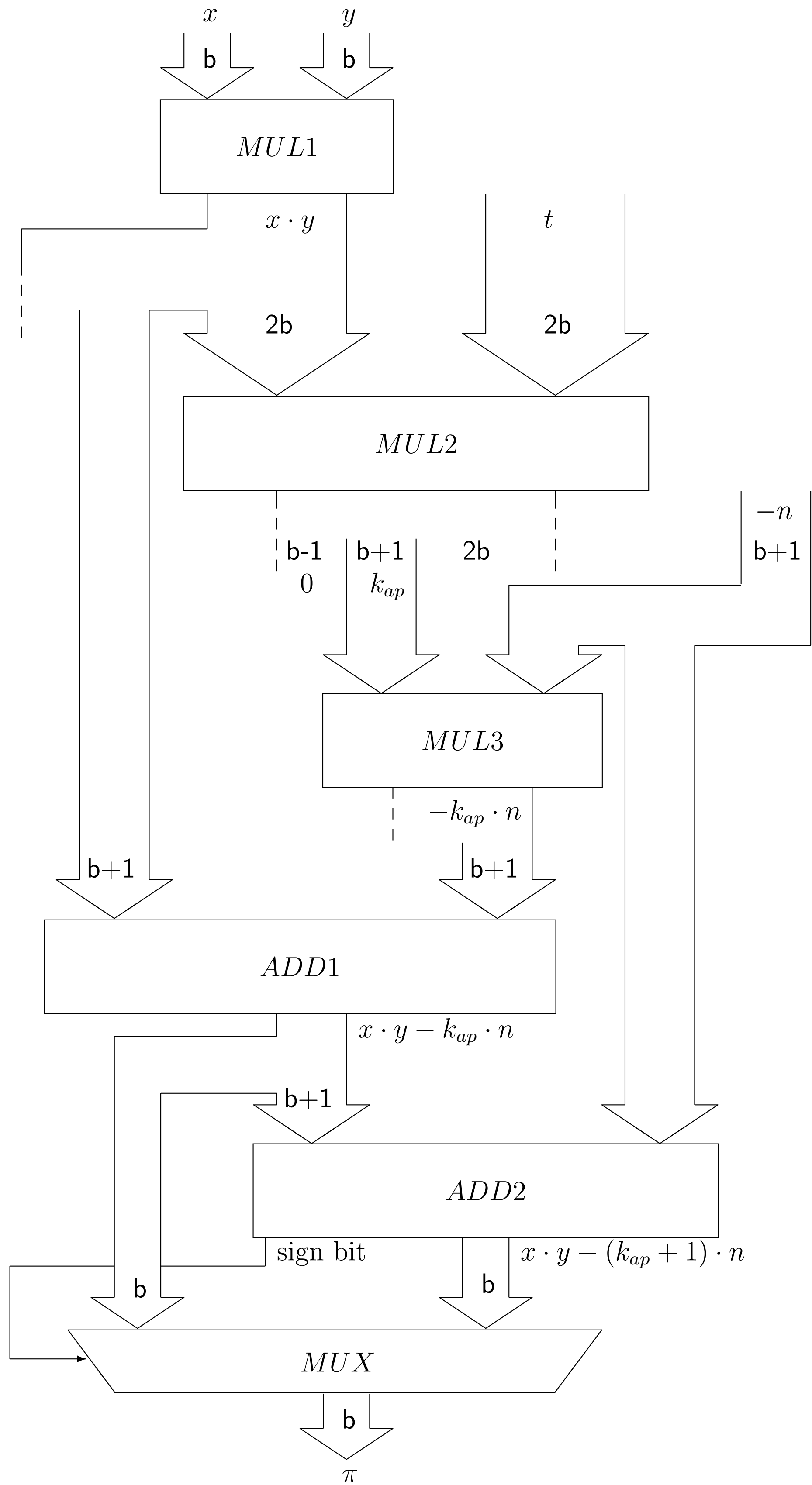


Fig. 1. Structure of the modulo n multiplier

$$S_i \leftrightarrow A_i \oplus B_i \oplus C_i \qquad i = 0 \ldots b-1$$

$$C_{i+1} \leftrightarrow ((A_i \wedge B_i) \vee (A_i \wedge C_i) \vee (B_i \wedge C_i)) \qquad i = 0 \ldots b-1$$

$$\sim C_0$$

Fig. 2. Boolean equations of a ripple-carry adder for b-bits

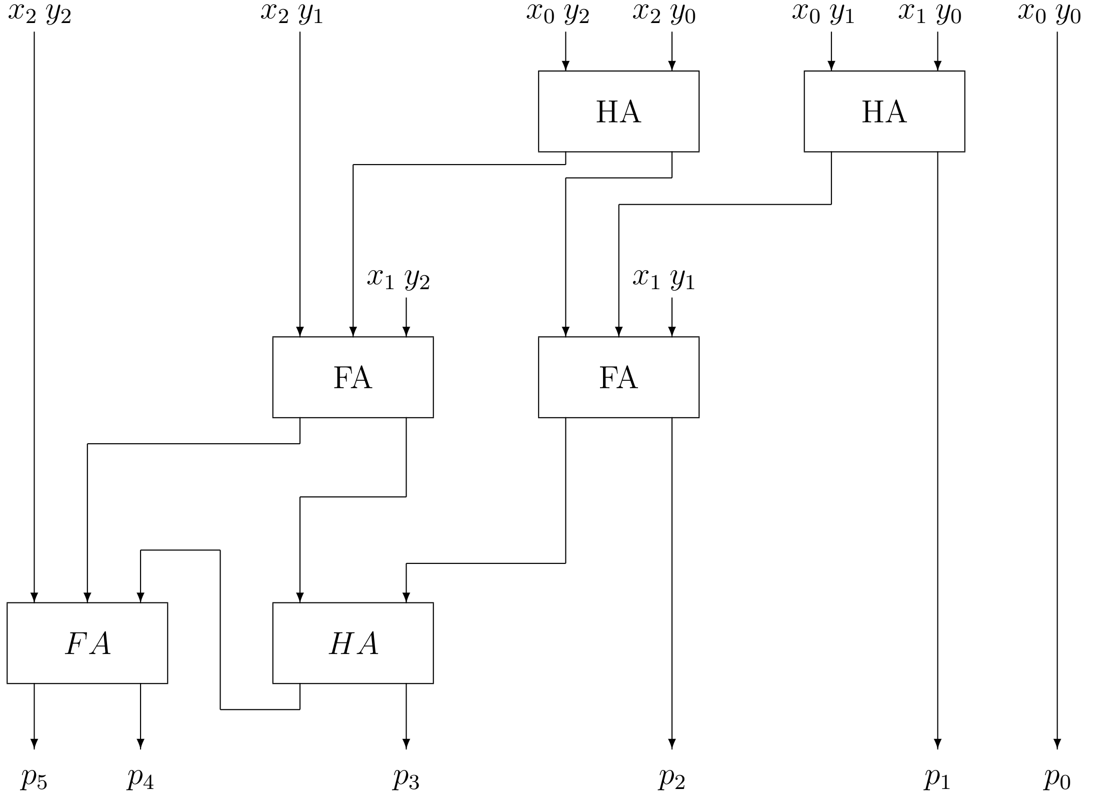


Fig. 3. An $b \times b$ array multiplier for $b = 3$

where m' is just the result of $f^2 \bmod n$.

5 Generating Satisfiable and Unsatisfiable Instances

This encoding makes it possible to generate both satisfiable and unsatisfiable SAT instances.

The simplest way to generate *solved satisfiable instances* is to use the SAT solver to search for *fake RSA signatures* according the following procedure:

(1) randomly generate a public key $\langle e, n \rangle$;
  (a) randomly generate a signature f;

$I_{i,j} \leftrightarrow X_i \wedge Y_j$	$i,j = 0 \ldots b-1$
$S_{0,j} \leftrightarrow I_{0,j+1} \oplus I_{j+1,0}$	$j = 0 \ldots b-2$
$S_{i+1,j} \leftrightarrow C_{i,j} \oplus S_{i,j+1} \oplus I_{j+1,i+1}$	$i,j = 0 \ldots b-2$
$C_{0,j} \leftrightarrow I_{0,j+1} \wedge I_{j+1,0}$	$j = 0 \ldots b-2$
$C_{i+1,j} \leftrightarrow ((I_{j+1,i+1} \wedge C_{i,j}) \vee (I_{j+1,i+1} \wedge S_{i,j+1}) \vee (C_{i,j} \wedge S_{i,j+1}))$	$i,j = 0 \ldots b-2$
$P_0 \leftrightarrow I_{0,0}$	
$P_i \leftrightarrow S_{i-1,0}$	$i = 1 \ldots b-1$
$P_{i+b} \leftrightarrow S_{b-1,i}$	$i = 0 \ldots b-2$
$P_{2b-1} \leftrightarrow C_{b-1,b-2}$	

Fig. 4. Boolean equations of an array multiplier

(b) compute $m = f^e \bmod n$;
(c) transform the numbers m, f, and n into the corresponding boolean values $\mathbf{v}_m$, $\mathbf{v}_f$ $\mathbf{v}_e$, $\mathbf{v}_n$;
(d) substitute in $\mathcal{R}SA_e(\mathbf{f}, \mathbf{m}, \mathbf{n})$ the corresponding boolean values that we have so far generated but for f.

The pair $\langle \mathbf{v}_f, \mathcal{R}SA_e(\mathbf{f}, \mathbf{v}_m, \mathbf{v}_n)\rangle$ gives a solved instance of the satisfiability problem. Since RSA was designed to be hard to break, this will provide us with the hard solved instances asked for by Cook and Mitchell [10].

If we want to generate just *satisfiable instances* we skip step 1a and replace step 1b with the following:

1b' randomly generate a message m;

The formula $\mathcal{R}SA_e(\mathbf{f}, \mathbf{v}_m, \mathbf{v}_n)$ is a satisfiable instance of the satisfiability problem. If we fix also n, then we can generate an inexaustible number of similar instances just by concatenating the randomly generated unit clauses corresponding to the description of $\mathbf{v}_m$ with the (constant) formula $\mathcal{R}SA_e(\mathbf{f}, \mathbf{m}, \mathbf{v}_n)$.

However, there is no way to generate an unsatisfiable formula by just changing f and m if e and n are chosen according Definition 1. Indeed, the condition that e is co-prime with $\phi(n)$ ensures that the equation $m = f^e \bmod n$ always has a solution. Whereas this is desired for the RSA cryptosystem [11], it is a bit annoying if we are interested in the use of RSA to generate hard SAT benchmarks.

However, we do not need any modification to the encoding to generate both

[11] This properties simply guarantees that every message can be decrypted.

satisfiable *and* unsatisfiable instances. We simply need to change the benchmark generation as follows:

(1) randomly generate a public key $\langle e, n \rangle$ where e violates Definition 1 and divides $\phi(n)$, i.e. e divides either $p-1$ or $q-1$ if $n = pq$;
   (a) randomly generate a message m;
   (b) transform the numbers m, f, and n into the corresponding boolean values $\mathbf{v}_m$, $\mathbf{v}_f$ $\mathbf{v}_e$, $\mathbf{v}_n$;
   (c) substitute in $\mathcal{R}SA_e(\mathbf{f}, \mathbf{m}, \mathbf{n})$ the corresponding boolean values that we have so far generated but for f.

If e divides $\phi(n)$ the equation $m = f^e \bmod n$ may not have a solution, i.e. *the formula* $\mathcal{R}SA_e(\mathbf{f}, \mathbf{v}_m, \mathbf{v}_n)$ *is satisfiable iff* m *is an* e*-th residue modulo* n. This problem is also hard, and substantially equivalent to the original RSA problem, and thus we have a general way to generate both satisfiable and unsatisfiable numbers.

An intriguing observation is that in number theory there is no way to show a proof that a number m is not an e-th residue [12] modulo n: we can only show a proof that a number is a residue by exhibiting the solution f. Here, a resolution proof of the unsatisfiability of $\mathcal{R}SA_e(\mathbf{f}, \mathbf{v}_m, \mathbf{v}_n)$ gives the desired proof that m is not an e-th residue.

6 Experimental Analysis

To automate the benchmark generation, we have designed and implemented a program to generate the encoding $\mathcal{R}SA_e(\mathbf{f}, \mathbf{m}, \mathbf{v}_n)$ where the modulus n can be a number of arbitrary size (i.e. possibly larger than the current C implementation of integers using 32/64 bits).

6.1 The Experimental Setting

For our experiments we have generated both solved instances and sat/unsat instances, according the methodology that we have presented in Section 5. The building blocks shared by both methods are schematized in Fig. 5.

The elements represented in figure 5 are the followings:

the random generator generates the pair of prime numbers p and q, factors of the modulus n, and possibly the signature f;

[12] In constrast we can exhibit a short proof that a number is prime [9].

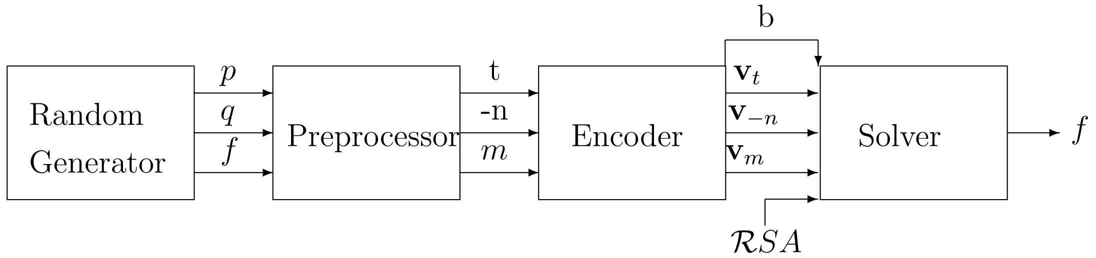


Fig. 5. Scheme of the Experimental Analysis

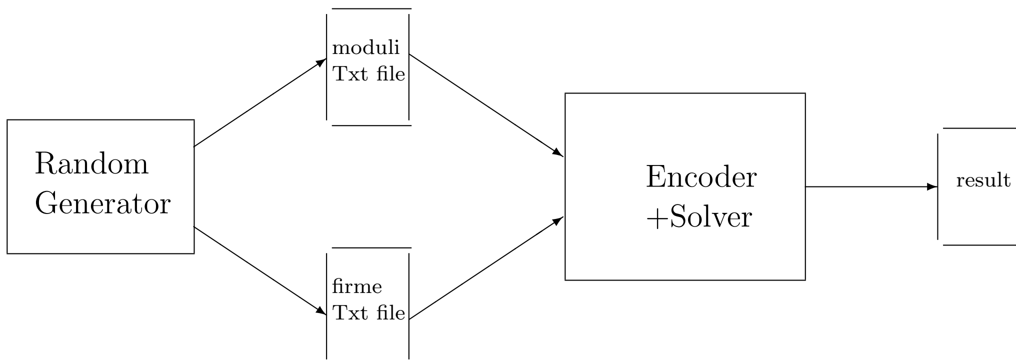


Fig. 6. Transformation of the scheme

the preprocessor processes the input data (p, q and f) in order to calculate the binary output data where $-n$ is the opposite in two-complement of the modulus, t is approximated value of $1/n$, m is the message obtained from the modular product $f^3 \bmod n$ (if solved instances are sought);

the encoder encodes the input data in the format of the system and sends them in output with b, that symbolizes the number of bits of n;

the system uses the formula $\mathcal{R}SA_3(\mathbf{f}, \mathbf{v}_m, \mathbf{v}_n)$ and the other input data to search a model that satisfies the input formula.

We slightly modified the above schema to make experiments reproducible i.e. we have stored at least some of the moduli and some of the signatures. Thus, figure 6 shows the transformation of the scheme in figure 5.

For the "random generator" and the "preprocessor", we have used the software package LIP by A. Lenstra [26], containing a variety of functions for arithmetic on arbitrarily large integers.

The word "Solver" denotes eqsatz [27], HeerHugo [19], smodels [31] and other systems we have tested. HeerHugo is a saturation base procedure which uses a variant of the Stålmark algorithm based on clauses, and eqsatz is a variant

Let Σ be a set of clauses using variables in V and k an integer

- Apply some clause-based simplification rules to Σ (unit, subsumption, restricted resolution etc.);
- If Σ contains an empty clause return UNSAT;
- If Σ contains only literals return SAT;
- If $k = 0$ return Σ;
- (Branch Rule) Select a variable v in V,
  - · assign v a truth value and call recursively HeerHugo on the simplified set of clauses with $k-1$ and let Σ_1 be the result;
  - · Assign v the opposite truth value and call recursively HeerHugo on the simplified set of clauses with $k-1$ and let Σ_2 be the result;
  - · (Merge Rule) If either of Σ_i is SAT then set $\Sigma = \Sigma_i$, else if either of Σ_i is UNSAT then set $\Sigma = \Sigma_j$ with $j \neq i$, otherwise put $\Sigma = \Sigma_1 \cap \Sigma_2$ and restart.

Fig. 7. The Saturation Method by HeerHugo

of DPLL which include equational reasoning for dealing with exclusive or, smodels is an efficient DPLL implementation of the stable model semantics of logic programs that has some features in common with HeerHugo.

The core algorithm for HeerHugo is sketched in Fig. 7. For any given formula, there is a value of k for which the formula is either proved unsatisfiable or satisfiable. Thus search is simply a form of iterative deepening in which the algorithm is called with increasing values of k until a solution is found. For further details see [19].

The core algorithm for eqsatz is sketched in Fig. 8. It is a fast implementation of the classical DPLL branching algorithm [14] enhanced with a special subroutine that recognizes subset of clauses representing affine formulae (i.e. formulae representable with exclusive or as the only connective) and applies specialized rules to that subset. For further details see [27].

The core algorithm for smodels is sketched in Fig. 9. It applies the classical DPLL branching algorithm [14] to logic programs with negation as failure and the stable model semantics. The advantage of logic programs is that we must only specify positive rules, i.e. rules to make variables true, since everything else is false by default. As a result, the size of the input problem halves wrt a clausal representation. The program also has a lookahead step which is fairly similar to the merge rule of HeerHugo. For further details see [30,31].

On top of smodels we have added a preprocessing step that is already incorporated in HeerHugoand eqsatz: substantially we apply various forms of unit propagation and simplification to make the formula smaller before doing any actual search. For further details on the simplifier see [17].

Let Σ be a set of clauses using variables in V.
- if Σ is empty return SAT;
- if Σ contains an empty clause return UNSAT;
- (Unit Propagation) while Σ contains a unit clause $\{l\}$ then assign l the value true and simplify Σ;
- (Equivalency Reasoning) Simplify Σ with the ad-hoc rules for reasoning about clauses corresponding to formulae with exclusive or;
- (Splitting Rule) Select a variable v in V, assign v a truth value and call recursively eqsatz on the simplified set of clauses. If eqsatz returns SAT then return SAT, otherwise assign v the opposite truth value and return the result of eqsatz on the simplified set of clauses.

Fig. 8. The DPLL Method by eqsatz

Let Π be a set of logic programming rules representing the clauses in Σ using variables in V.
- if Π is empty return SAT;
- if Π contains an empty clause return UNSAT;
- (Unit Propagation) while Π contains a unit clause $\{l\}$ (a fact) then assign l the value true and simplify Π;
- (Lookahead) For all variables v in V,
  - · assign v a truth value and apply Unit Propagation on the simplified logic program, if the result is UNSAT, assign v the opposite truth value;
  - · assign v the opposite truth value and apply Unit Propagation on the simplified logic program, if the result is UNSAT, assign v the first truth value;
- (Splitting Rule) Select a variable v in V, assign v a truth value and call recursively smodels on the simplified logic program. If smodels returns SAT then return SAT, otherwise assign v the opposite truth value and return the result of smodels on the simplified logic program.

Fig. 9. The DPLL Method by smodels

To check how SAT algorithm scaled wrt classical algorithm for computing cube roots we have also run a parallel test using Pollard-ρ method as the underlying factorization method. This control algorithm is sketched in Fig. 10. We have not used more advanced algorithms (such as the Elliptic Curves Method or the General Number Field Sieve) because they are competitive only when the number of decimal digits of the modulus n is over 10, far too large a number for our limited hardware and for our SAT Solvers.

Let m and n be integer with $m < n$
- factor n into its factors p and q using Pollard-ρ method.
- compute $\phi(n) = (p-1) \cdot (q-1)$
- compute d such that $3 \cdot d = 1 \bmod \phi(n)$ using Euclid's algorithm
- compute $f = m^d \bmod n$ using a standard algorithm for modular exponentiation.

Fig. 10. The Reference Pollard-ρ Method

The experiments were run on an Alpha with 256 MB of memory, a Pentium II with 64 MB of memory, and a Pentium III with 512 MB of memory. All computers run Linux as the operating system. No run was timed-out.

We have not reported the sizes of the instances in a table since they can be exactly calculated from the structure shown in Fig. 1 and they are around $O(6 \log^2 n)$. To give a feeling of the orders of magnitude, the encoding for a 22-bits modulus, after compacting, unit propagation and applying the unary failed-literal rule [12] has 40.016 clauses and over 7.000 variables. The RSA-129 challenge given by Martin Gardner in *Scientific American* in 1977 (see [47, pag.320] for a more recent reference) is encoded with 6.604.076 formulae (before preprocessing).

In contrast with the encoding of the state-of-the-art version of DES, which takes a "paltry" 60.000 clauses and nonetheless is hard to solve [28] , the number of clauses is much bigger in this case.

6.2 Generation of Solved Instances (Faking RSA Signature)

For solved instances the generation of the benchmark suite worked as follows:

(1) fix the number of bits of the RSA modulus we are interested in;
(2) randomly generate a modulus n as the product of two random primes;
(3) randomly generate 50 signatures
(4) for each generated signature f do
   (a) apply the modular exponentiation algorithm using the RSA public key $\langle e, n \rangle$ and generate the message $m = f^e \bmod n$ (here $e = 3$);
   (b) encode the modular exponentiation algorithm as $\mathcal{RSA}_e(\mathbf{f}, \mathbf{m}, \mathbf{n})$ and substitute the values of the message $\mathbf{v}_m$ and the modulus $\mathbf{v}_n$;
   (c) search for a model of the formula using a SAT solver.

If at step 2 we do not check whether $e = 3$ divides $\phi(n)$, the message m must be computed as the e-th power of a given f. Otherwise the formula may not be satisfiable. To test for correctness we always check that what we found out

Table 1
Sample of Results of the Faking RSA signature experiments

MOD.	BIT	SIGN.	SMODELS	HEER.	SATZ	POL.
6	3	4	0.02s	2.00s	0.11s	0.01s
6	3	5	0.00s	2.00s	0.09s	0.01s
49	6	9	0.42s	45.00s	0.44	0.02s
49	6	48	0.23s	45.00s	0.44	0.01s
143	8	104	0.71s	2915s	2.06s	0.07s
143	8	123	1.57s	3321s	2.07s	0.07s
667	10	128	0.89s	5h 2012s	19.12s	0.09s
667	10	276	5.43s	1d 20h 3207s	19.13s	0.09s
2,501	12	96	30.09s	3h 3090s	2.48s	0.13s
2,501	12	1,259	21.11s	3d 04h 2943s	2.45s	0.13s
7,597	13	497	229.80s	4d 01h 3090s	5.89	0.16s
7,597	13	7,258	190.37s	3d 16h 2002s	7.00	0.16s
29,213	15	8,304	20.67s	11d 04h 0620s	4.92s	0.19s
29,213	15	27,704	491.65s	5d 07h 2291s	31.39s	0.19s
156,263	18	80,465	1189s	7d 10h 3388s	8.32s	0.21s
156,263	18	53,477	1h 2699s	6d 13h 1099s	11.05s	0.21s
455,369	19	84,882	8h 3578s	10d 16h 2118s	1204s	0.23s
455,369	19	405,346	1h 1948s	10d 05h 3482s	923s	0.23s
2,923,801	22	1,847,296	20h 0311s	–	1423s	0.25s
2,923,801	22	1,983,121	11h 3500s	–	2d 11h 2708s	0.25s
13,340,267	24	3,958,651	14h 0598s	–	–	–
13,340,267	24	11,376,425	7h 0467s	–	–	–
28,049,353	25	18,67,233	1d 03h 1744s	–	–	–
28,049,353	25	20,910,282	19h 1934s	–	–	–
183,681,697	28	92,751,060	17h 2721s	–	–	–
183,681,697	28	114,808,473	1d 12h 0077s	–	–	–
504,475,141	29	301,368,039	1d 03h 1346s	–	–	–
504,475,141	29	273,472,864	2d 08h 2055s	–	–	–

was indeed a correct signature.

If a model exists we have found a cubic root of m modulo n, i.e. we have been able to fake the RSA signature of message m for the public key $\langle 3, n \rangle$.

In Table 1 we show a sample of the results, to give a feeling of the orders of magnitudes of running times of different solvers. We report

- the modulus,
- the number of bits of the modulus,
- the signature,
- and for each system the running time (in seconds, hours and days).

HeerHugo showed the worst performance and we had to stop the experiments when they required more than 10 days to solve one instance. eqsatz was by far the fastest, if the formula is sufficiently small to fit into the processor's cache [13]. As soon as the size of the problem increases beyond that point its performance is no longer predictable. Thus we decided to use smodels which offered a good compromise between speed and stability of performance for larger moduli.

To measure the scalability of the SAT-solvers we have used the methodology proposed by Fleming and Wallace [18], as we have run the experiments on different machines and at different times. This means that for every size of the modulus (in bits) we compute the geometric mean of the running time and then divide that mean time by the geometric mean time that the system has taken on the smallest modulus.

In a nutshell, the time for computing a cube root modulo a 3-bit number is the reference problem and all other values are normalized by that number. For instance, using the data in Table 1 for HeerHugo we would have:

$$\begin{aligned}
t_{3bits} &= \sqrt{2.00s \cdot 2.00s} &&= 2.00s \\
t_{6bits} &= \sqrt{45.00s \cdot 45.00s} &&= 45.00s \\
t_{8bits} &= \sqrt{2915s \cdot 3321s} &&= 3111.38s \\
Nt_{3bits} &= \frac{t_{3bits}}{t_{3bits}} = \frac{2.00s}{2.00s} &&= 1.00 \\
Nt_{6bits} &= \frac{t_{6bits}}{t_{3bits}} = \frac{45.00s}{2.00s} &&= 22.50 \\
Nt_{8bits} &= \frac{t_{8bits}}{t_{3bits}} = \frac{3111.38s}{2.00s} &&= 1555.69
\end{aligned}$$

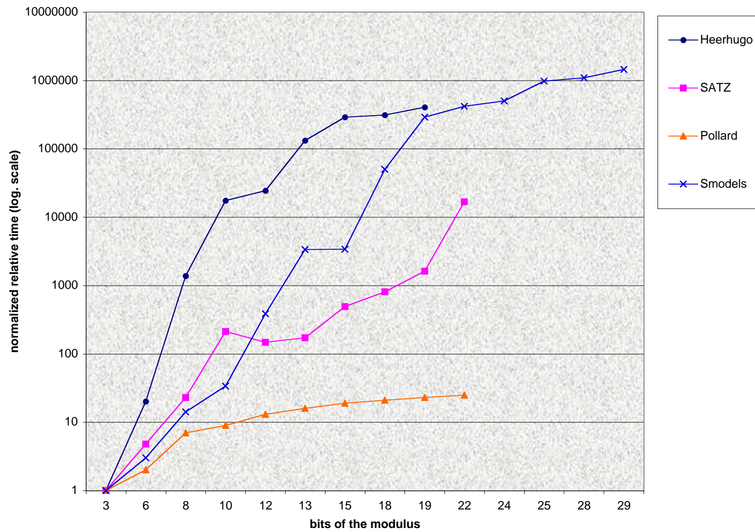
Notice that the final value is not a dimensional measure (i.e. seconds, hours etc.) but just a comparative indication. In other words, $Nt_{8bits} = 1555.69$ means that HeerHugo runs 1555 times slower on moduli of 8 bits than it runs on moduli of 3 bits.

The results are reported in Fig. 11.

The flattening of the curves on the logarithmic scale shown by smodels, HeerHugo, and Pollard (a factoring algorithm), is consistent with the worst case complexity of factoring algorithms, namely a subexponential [14] algorithm in

[13] Retrospectively, this was to be expected as satz, the core of eqsatz without rules for exclusive or, has been optimized for running on randomly generated CNF at the cross-over point and thus for around few hundred variables and around few thousand clauses.

[14] Recall that on a logarithmic scale an exponential algorithm is mapped into a straight line.



.

Fig. 11. Comparison of Heerhugo-SATZ-Pollard Normalized Running Times

the number of bits of the modulus.

6.3 Generation of Sat/Unsat Instances (Solving Cubic Residuosity)

To analyze the relative hardness of satisfiable/unsatisfiable instances, we followed the systematic approach used in the analysis of Random-CNF [39]. In our setting, this boils down to the following algorithm:

(1) fix the number of bits of the RSA modulus we are interested in;
(2) generate randomly a modulus n as the product of two random primes of the appropriate size such that $e = 3$ divides $\phi(n)$ is we want both satisfiable and unsatisfiable instances;
(3) for all possible values of $m = 0 \dots n-1$ do
    (a) encode the modular exponentiation algorithm as $\mathcal{R}SA_e(\mathbf{f}, \mathbf{m}, \mathbf{n})$ and substitute the values of the message $\mathbf{v}_m$ and the modulus $\mathbf{v}_n$;
    (b) search for a model of the formula using a SAT solver.

If a model exists we have found a cubic root of the value m modulo n. If the formula is unsatisfiable we showed that m is a cubic non-residue modulo n.

Table 2
Sample of Results for Solving Cubic Residuosity

MOD.	MSG	SIGN.	CHOICE	TIME
35	0	0	0	0.170
35	1	16	0	0.200
35	2	n.r.	14	0.690
35	3	n.r.	22	0.840
35	4	n.r.	19	0.840
35	5	n.r.	21	0.720
35	6	26	1	0.270
35	7	28	5	0.370
35	8	32	0	0.180
35	9	n.r.	18	0.750
35	10	n.r.	14	0.700
35	11	n.r.	20	0.820
35	12	n.r.	22	0.880
35	13	17	0	0.310
35	14	14	7	0.500
35	15	25	1	0.260
35	16	n.r.	18	0.780
35	17	n.r.	20	0.790

MOD.	MSG	SIGN.	CHOICE	TIME
35	18	n.r.	14	0.660
35	19	n.r.	21	0.820
35	20	20	1	0.230
35	21	21	2	0.240
35	22	18	0	0.210
35	23	n.r.	20	0.880
35	24	n.r.	19	0.840
35	25	n.r.	18	0.880
35	26	n.r.	14	0.690
35	27	13	1	0.220
35	28	7	5	0.390
35	29	4	2	0.210
35	30	n.r.	14	0.680
35	31	n.r.	20	0.870
35	32	n.r.	14	0.710
35	33	n.r.	14	0.690
35	34	24	0	0.220

Note that a systematic approach is necessary because we want an indication of the relative difficulty of the sat/unsat cases and there is no general (and efficient) algorithm to provably generate residue and non-residues modulo n. The same procedure is followed by Selman et al. [39] to explore the hardness of the Random-CNF SAT benchmark. Clearly, to determine whether a particular m is a cubic non-residue there is no need of generating all possible values from 1 to n: it is enough to test the satisfiability of the single formula $\mathcal{R}SA_e(\mathbf{f}, \mathbf{m}, \mathbf{n})$ with the values of the message $\mathbf{v}_m$ and the modulus $\mathbf{v}_n$. To generate benchmarks with larger formulae, when a systematic sweep of the search space is no longer possible, the messages m can be generated at random.

On this benchmark we have run only the smodels system because it offered the best compromise between speed and predictability of running time (i.e. for the same modulus we have not found an instance requiring 2 days and another requiring 30 minutes as we had with eqsatz).

In Table 2 we show a sample of the results, to give an indication of the order of magnitude of running times on the entire search space for a small modulus $35 = 5 \cdot 7$. Here, we also indicate the number of choice points and the running time in seconds. "n.r" means that the number in the column MSG is a non-residue modulo 35.

Notice how satisfiable and unsatifiable instances are well distributed and that satisfiable instances require practically no choices. The solver is sufficiently

Table 3
Quantitative Results for Solving Cubic Residuosity

Results when the modulus is a small prime							
All instances				SAT instances		UNSAT instances	
Bits	Instances	%SAT	%UNSAT	Choices	Time	Choices	Time
3	12	67%	33%	1	0.04	2	0.04
4	24	67%	33%	2	0.08	5	0.13
5	119	73%	27%	4	0.19	9	0.23
6	37	35%	65%	5	0.27	21	0.79

Results when the modulus is the product of two small primes							
All instances				SAT instances		UNSAT instances	
Bits	Instances	%SAT	%UNSAT	Choices	Time	Choices	Time
6	90	78%	22%	9	0.43	18	0.78
7	815	50%	50%	14	0.96	40	1.99
8	2242	51%	49%	25	2.59	75	5.57
9	4271	50%	50%	50	6.09	153	15.66
10	2999	26%	74%	65	12.49	297	49.02

smart to find the solution without substantial search.

In Table 3 we show more quantitative data, namely average running time for satisfiable and unsatisfiable instances. The first table shows the result when the modulus n is a prime of size up to 6 bits (an easy problem in number theory) and the second table shows the results when n is composite, i.e. the product of two such primes.

In the table we report:

- the number of bits of the modulus,
- the number of tested instances for all moduli with that size,
- the relative percentage of SAT/UNSAT instances
- and for each type of instances (SAT/UNSAT) the average number of choice points and the average running time in seconds.

We have not given the overall average running time as the data can be substantially bi-partite in two regions, one for unsatisfiable formulae and one for satisfiable formulae.

In contrast with Random-CNF benchmarks [39], in this benchmark the hardness of the problem is unrelated to phenomena like phase transitions in the satisfiablity/unsatisfiability ratio of instances.

6.4 What the benchmark tells on different systems

The difference in performance between HeerHugo and eqsatz is significant because both systems have reduction rules beyond unit propagation that are able to cope with affine subformulae in quite effective ways and that are fairly similar. For instance both systems have rules to derive $p \leftrightarrow q$ and then replace q with p everywhere in the formula.

Apparently HeerHugo breadth first search system is not very effective unless a proof of unsatisfiability can be easily found. In all other cases, the memory requirement of the procedure and the necessity of a substantially exhaustive case analysis of many control variables (the branch-merge phase for larger k) before a solution is found makes the procedure not suitable. In contrast eqsatz, even if its underlying proof system is less strong than the proof system of HeerHugo, can relatively quickly head for a solution. An (expected) consequence of this difference in the search procedure is that HeerHugo is much more reliable than eqsatz for what regard the variance of running times.

The benchmark was also good in pointing the "short-lived" nature of the optimized coding of the data structures of eqsatz: on larger benchmarks, when the problem instance could no longer fit into the processor's cache, the performance of the solver becomes unpredictable. Most likely, this large variance in performance can be explained by the need of swapping data to-and-from the main memory.

What is more significant is the relatively good performance of smodels whose proof system is definitely weaker than both HeerHugoand eqsatz. Apparently, only the unsatisfiable part of the merge rules in HeerHugo seems to pay off in these examples.

7 Conclusions

In this paper, we have shown how to encode the problem of finding (i.e. faking) an RSA signature for a given message without factoring the modulus. This corresponds to computing the e-th root modulo n (e-residuosity modulo n). In the number theory field no solution to this problem is known, when n is not a prime, without a preliminary step equivalent to factoring n.

We have shown how the encoding of the e-th residuosity modulo n problem can be used for generating solved instances and satisfiable and unsatisfiable instances. Our encoding extends the set of number theoretic problem that can be used to generate SAT-benchmarks over the initial proposal by Cook and Mitchell [10]: factoring large integers by encoding the problem into SAT and using SAT-Solvers.

We believe that using such cryptographic benchmark can be beneficial for SAT Research as they combine into one framework the properties of structured problems and randomly generated problems and namely:

- they require a rich set of connectives which makes it possible to test formulae beyond CNF;
- they are structured as typically happens for formulae coming from real world applications;
- problem instances can be randomly generated in almost inexhaustible numbers, by varying either the solution or the instance (while keeping the same solution) or both;
- we can control the nature of the instance (satisfiable or unsatisfiable) without making it too easy to solve;
- last but not least they are interesting problems on their own (and whose importance is much easier to grasp for the layman than the n-queen puzzle or the DIMACS parity problem).

The experiments on SAT provers, HeerHugo by Groote and Warners [19], eqsatz by Li [27], and smodels by Niemela and Simmons [30] shows that SAT Solvers are well behind number theoretic algorithms which solve the same problem using factoring but are not totally hopeless.

The first avenue of future research is the testing of other algorithms which are able to exploit the presence of affine subproblems even more than HeerHugo and eqsatz. Indeed, in contrast with the encoding of DES reported in [28], here the affine subproblem is almost 50% of the whole formula. A possible approach is to apply algorithms such as those by Warners and Van Maaren [46] as a pre-processing phase, another approach is to test the effectiveness of specialized calculi which integrate more closely affine and clausal logic such as those by Baumgartner and Massacci [4].

Finally, an intriguing path suggested by an anonymous reviewer is representing integers with Gray codes such that the bitwise represention of n and $n+1$ also differ by one bit. This might give SAT-solvers with lemmaizing a competitive edge on this kind of problems: lemmas could more easily cut off a large portion of the search space. However, the design of combinatorial arithmetic circuits (addition, multiplication and two-complement substraction) for these codes is a research problem in itself.

We leave these issues open for future investigations.

References

[1] G. Alia, E. Martinelli, A VLSI modulo m multiplier, IEEE Transactions on Computers 40 (7) (1991) 873–878.

[2] J. Bajard, L. Didier, P. Kornerup, An RNS Montgomery modular multiplication algorithm, IEEE Transactions on Computers 47 (7) (1998) 766–776.

[3] D. Balenson, Privacy enhancement for internet electronic mail: Part iii: Algorithms, modes and identifiers, Tech. Rep. RFC 1423, IETF (February 1993).

[4] P. Baumgartner, F. Massacci, The taming of the (X)OR, in: Lloyd et al. (Eds.), Computational Logic – CL 2000, First International Conference, Vol. 1861 of Lecture Notes in Artificial Intelligence, Springer-Verlag, 2000, pp. 508–522.

[5] D. Boneh, Twenty years of attacks on the RSA cryptosystem, Notices of the American Mathematical Society 46 (2) (1999) 203–213.

[6] D. Boneh, Venkatesan, Breaking RSA may not be equivalent to factoring, in: Advances in Cryptology - Eurocrypt 98, Vol. 1403 of Lecture Notes in Computer Science, Springer-Verlag, 1998, pp. 59–71.

[7] R. Bryant, Graph-based algorithms for boolean function manipulation, IEEE Transactions on Computers 35 (8) (1986) 677–691.

[8] L. J. Claesen (Ed.), Formal VLSI Correctness Verification: VLSI Design Methods, Vol. II, Elsevier Science Publishers (North-Holland), Amsterdam, 1990.

[9] H. Cohen, A Course in Computational Algebraic Number Theory, Springer-Verlag, 1995.

[10] S. Cook, D. Mitchel, Finding hard instances of the satisfiability problem: A survey, in: Du, Gu, Pardalos (Eds.), Satisfiability Problem: Theory and Applications, Vol. 35 of DIMACS Series in Discrete Mathamatics and Theoretical Computer Science, American Mathematical Society, 1997, pp. 1–17.

[11] D. Coppersmith, Small solutions to polynomial equations, and low exponent RSA vulnerabilities, Journal of Cryptology 10 (1997) 233–260.

[12] J. Crawford, L. Auton, Experimental results on the crossover point in random 3SAT, Artificial Intelligence 81 (1-2) (1996) 31–57.

[13] M. Davis, H. Putnam, A computing procedure for quantificational theory, Journal of the ACM 7 (3) (1960) 201–215.

[14] M. Davis, G. Longemann, D. Loveland, A machine program for theorem-proving, Communications of the ACM 5 (7) (1962) 394–397.

[15] M. Dugdale, Residue multipliers using factored decomposition, IEEE Trans. Circuits and Systems-II: Analog and Digital Signal Processing 41 (1994) 623–627.

[16] K. Elleithy, M. Bayoumi, A systolic architecture for modulo multiplication, IEEE Trans. Circuits and Systems-II: Analog and Digital Signal Processing 42 (1995) 725–729.

[17] C. Fiorini, Criptanalisi Logica di RSA, Master's thesis, Facoltà di Ingegneria, Univ. di Roma I "La Sapienza", in Italian (May 2000).

[18] P. Fleming, J. Wallace, How not to lie with statistics: the correct way to summarize benchmark results, Communications of the ACM 29 (3) (1986) 218–221.

[19] J. Groote, J. Warners, The propositional formula checker HeerHugo, Journal of Automated Reasoning 24 (1) (2000) 101-125.

[20] J. Harrison, Stalmarck's algorithm as a HOL derived rule, in: Proceedings of the Ninth International Conference on Theorem Proving in Higher Order Logics (TPHOLs'96), Vol. 1125 of Lecture Notes in Computer Science, Springer-Verlag, 1996, pp. 221–234.

[21] J. Hastad, Solving simultaneous modular equations of low degree, SIAM Journal on Computing 17 (1988) 336–341.

[22] A. Hiasat, New efficient structure for a modular multiplier for RNS, IEEE Transactions on Computers 49 (2) (2000) 170–174.

[23] G. Jullien, Implementation of multiplication, modulo a prime number, with applications to theoretic transforms, IEEE Transactions on Computers 29 (10) (1980) 899–905.

[24] ITU. Recommendation X.509: The Directory - Authentication Framework, (1988). Available on the web at URL `http://www.itu.int/itudoc/itu-t/rec/x/x500up/x509.html`

[25] A. Lenstra, K. Lenstra, M. Manasse, J. Pollard, The number field sieve, Vol. 1554 of Lecture Notes in Mathematics, Springer-Verlag, 1993, pp. 11–42.

[26] A. K. Lenstra, Documentation of LIP, version 0.5 Edition, available by ftp at `ftp.ox.ac.uk` (March 1995).

[27] C.-M. Li, Integrating equivalency reasoning into Davis-Putnam procedure, in: Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI 2000), AAAI Press/The MIT Press, 2000, pp. 291–296.

[28] F. Massacci, L. Marraro, Logical cryptanalysis as a SAT-problem: Encoding and analysis of the U.S. Data Encryption Standard, Journal of Automated Reasoning 24 (1-2) (2000) 165–203.

[29] D. Mitchell, H. Levesque, Some pitfalls for experimenters with random SAT, Artificial Intelligence 81 (1-2) (1996) 111–125.

[30] I. Niemelä, Logic programs with stable model semantics as a constraint programming paradigm, Annals of Mathematics and Artificial Intelligence 25 (3-4) (1999) 241–273.

[31] I. Niemelä, P. Simmons, Smodels – an implementation of Stable Model and Well-founded Semantics for Normal Logic Programs, in: Proceedings of the Fourth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97), Vol. 1265 of Lecture Notes in Artificial Intelligence, Springer-Verlag, 1997, pp. 420–429.

[32] D. Radhakrishnan, Y. Yuan, Novel approaches to the design of VLSI RNS multipliers, IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing 39 (1992) 52–57.

[33] A. Ramnarayan, Practical realization of mod p, p prime, multiplier, Electronics Letters 16 (1980) 466–467.

[34] H. te Riele, New factorization record, NMBRTHRY, NMBRTHRY@LISTSERV.NODAK.EDU (August 1999).

[35] R. Rivest, A. Shamir, L. Adleman, A method for obtaining digital signatures and public-key cryptosystems, Communications of the ACM 21 (2) (1978) 120–126.

[36] RSA Laboratories, RSA Security, PKCS-1: RSA Encryption Standard, 1st Edition (1993). - Available on the web at URL http://www.rsasecurity.com/rsalabs/pkcs

[37] B. Schneier, Applied Cryptography: Protocols, Algorithms, and Source Code in C, John Wiley & Sons, 1994.

[38] B. Selman, H. Kautz, D. McAllester, Ten challenges in propositional resoning and search, in: Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97), Morgan Kaufmann, Los Altos, 1997, pp. 50–54.

[39] B. Selman, D. Mitchell, H. Levesque, Generating hard satisfiability problems, Artificial Intelligence 81 (1-2) (1996) 17–29.

[40] J. Singer, I. Gent, A. Smaill, Backbone fragility causes the local search cost peak, Tech. Rep. APES-17-1999, APES Research Group - Univ. of St. Andrews (1999).

[41] R. Silverman, A cost-based security analysis of symmetric and asymmetric key lengths, RSA Bulletin 13, RSA Laboratories, (April 2000). Available on the web at URL http://www.rsasecurity.com/rsalabs/

[42] M. Soderstrand, C. Vernia, A high-speed low-cost modulo pi multiplier with RNS arithmetic application, Procedings of the IEEE 68 (1980) 529–532.

[43] D. R. Stinson, Cryptography: Theory and Practice, CRC Press, Inc., New York, 1995.

[44] F. J. Taylor, A VLSI residue arithmetic multiplier, IEEE Transactions on Computers 31 (6) (1982) 540–546.

[45] C. Walter, Systolic modular multiplier, IEEE Transactions on Computers 42 (3) (1993) 376–378.

[46] J. Warners, H. van Maaren, A two phase algorithm for solving a class of hard satisfiability problems, Operations Research Letters 23 (3-5) (1999) 81–88.

[47] S. Yan, Number Theory for Computing, Springer-Verlag, 1999.