Encoding Basic Arithmetic Operations for SAT-Solvers

Ramón BÉJAR [1], Cèsar FERNÁNDEZ and Francesc GUITART
Computer Science Department, Universitat de Lleida (UdL)

Abstract. In this paper we start an investigation to check the best we can do with SAT encodings for solving two important hard arithmetic problems, integer factorization and discrete logarithm. Given the current success of using SAT encodings for solving problems with linear arithmetic constraints, studying the suitability of SAT for solving non-linear arithmetic problems was a natural step. However, our results indicate that these two problems are extremely hard for state-of-the-art SAT solvers, so they are good benchmarks for the research community interested in finding good SAT encodings for practical constraints.

Keywords. Satisfiability benchmarks, cryptography, arithmetic operations.

Introduction

The satisfiability problem (SAT) is the problem of determining whether there exists a satisfying assignment for a conjunctive normal form formula (CNF). The current high efficiency of SAT-Solvers turned SAT encodings a powerful tool for many practical industrial applications, such as Electronic Design Automation (EDA) and important problems in Artificial Intelligence, like STRIPS planning [7,8], that were originally believed to be problems not suitable for propositional logic satisfiability algorithms.

Given the current success of using SAT encodings for solving problems with linear arithmetic constraints, see for example [3], in this work we start an investigation for finding the best we can do with SAT encodings for solving non-linear arithmetic problems. We consider two such problems, integer factorization and discrete logarithm over a finite cyclic group, that are basic problems for cryptographic applications. These two problems are also interesting from the point of view of artificial intelligence and constraint programming, as they are problems in the complexity class NP, but that even if they are believed not to be polynomially solvable, they seem not to be NP-complete, so they are one of the few natural problems to be located somewhere between P and NP-complete problems, a class of problems not widely studied by these research communities. We present SAT encodings for the basic building functions that can be used to define these problems: adders and multipliers. Our SAT encodings are based on Boolean circuit representations of such functions, following the best approaches found so far for encoding linear constraints with SAT. For the multiplication function, we also consider a translation to pseudo-Boolean linear equations, and a subsequent transformation to SAT, using current SAT encodings for such equations. We compare the performance of our

[1] Corresponding Author eMail: ramon@diei.udl.cat.

SAT-based approaches with the best algorithms for such problems: Quadratic Sieve factorisation and Pollard's Rho discrete logarithm algorithms. Our results indicate that the performance of the SAT encodings are worse than for the current best specialized algorithms, thus indicating that these problems are interesting benchmarks for discovering efficient SAT encodings of practical constraints.

Nevertheless, we observe that the gap at performance between SAT encodings and specialized algorithms becomes narrower for discrete logarithm problems than for factorisation. This opens a new line of research towards elliptic curve cryptography, where we expect to narrow the gap, given the state-of-the-art of specialized elliptic curve algorithms.

1. Integer Factorization

The decomposition of an integer $n = p_1^{e_1} \cdot p_2^{e_2} \ldots p_k^{e_k}$, being p_i primes and e_i naturals, is unique. When we talk about n factorization we mean the problem of finding a non trivial decomposition for n. In this work we consider the special case of finding the non-trivial decompositions for an integer n that is the product of two primes x and y, where $x > 1$ and $y > 1$.

Factorization is one of the two cornerstones where RSA security relies on. RSA [12] is the first public-key cryptography schema suitable for digital signature as well as for encryption, being still widely used today. RSA security is based on the computational unfeasibility to factorize a public large integer, $n = x \cdot y$, where their prime factors, x and y, require certain conditions in order to avoid specific attacks that could discover x and y easily. Practical RSA public keys, nowadays consist of integers n with a length of a few thousands of bits.

In order to proceed with a SAT encoding for the factorization problem, we first need to define the basic building blocks, such as adders and multipliers. We will need adders to perform multiplications. Our first encoding will be based on a circuit build as an array of full and half adders. Later, in Booth multiplier we will also need to perform a carry look-ahead addition where adders will be necessary. Further we will use multipliers as modules, for exponentiation and multiplication in $\mathbb{Z}_n$.

1.1. Adders

Binary addition is a simple operation between two binary numbers. The simplest case is for inputs x and y with one bit length, so addition is defined with the four elemental equations: $0 + 0 = 00$, $0 + 1 = 01$, $1 + 0 = 01$ and $1 + 1 = 10$. So, the output is formed by two digits, the rightmost is the sum result (S) and the leftmost (C) is called Carry. We use X and Y as inputs.

1.1.1. Half Adder

A half adder needs two input bits and two output bits. The Carry bit is set to 1 only when the sum cannot be represented with a single bit. In Figure 1 the propositional logic representation of the equations that link the input with the output are shown.

$S \leftrightarrow X \oplus Y$ $C \leftrightarrow X \wedge Y$	$S \leftrightarrow X \oplus Y \oplus Z$ $C \leftrightarrow (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z)$

Figure 1. Left box: Boolean equations for a Half Adder; Right box: Boolean equations for a Full Adder

1.1.2. Full Adder

A Full Adder is a combinational circuit that forms the arithmetic sum of three input bits X, Y and Z. As the sum can have any value between 0 and 3, two output bits are needed as before but now all the four possible results can be obtained depending on the input values. In Figure 1 the propositional logic equations for a Full Adder are shown. S and C will be our outputs again.

1.2. Multipliers

1.2.1. Array Multiplier

For this implementation we used the propositional logic model proposed in [5], represented in Figure 2. It can be implemented as an array of half and full adders. Using this schema, two l-bits long numbers can be multiplied using an array with l rows and $2l$-1 columns. For the encoding, as well as for Figure 3 we use X and Y as input variables and P as output. Figure 3 shows the array multiplier for $l = 4$.

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

Figure 2. Boolean equations of an array multiplier

1.2.2. Booth Multiplier

Booth multiplication is a technique whereby x and y may be multiplied following a few simple steps [2]. We decided to implement such algorithm because the corresponding circuit representation is smaller than for the array multiplier in terms of size.

The Booth multiplier works with an iterative process, where the number of iterations is determined by the size of the multiplicand y, and where a sum of partial products is iteratively updated until it finally contains the value of the multiplication. First of all, let's define our variables. The input variables are the bit vectors multiplicands X, of size l_x, and Y, of size l_y, which contains the binary representation of x and y respectively. We define the size of the output as $l = l_x + l_y + 1$, and define the following additional bit vectors:

1. A is a bit vector of size l and contains the value of x on its l_x left-most bits, being the remaining $(l_y + 1)$ bits filled with zeros.

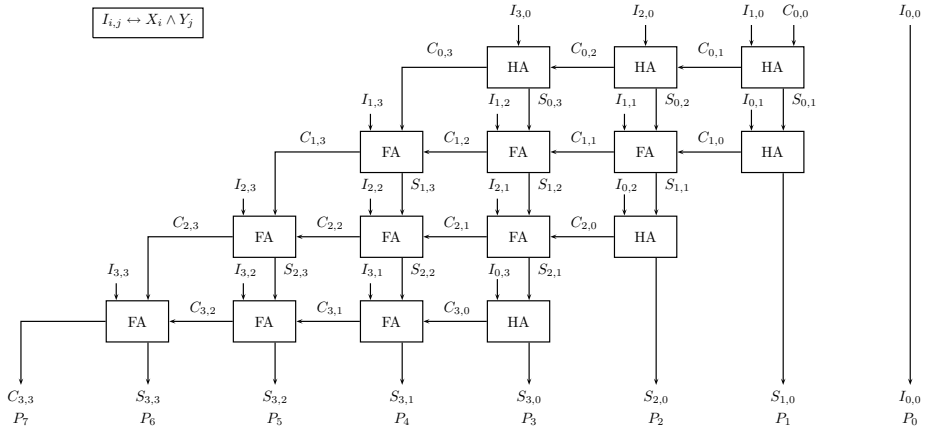

Figure 3. Modular design of an array multiplier.

2. S is a bit vector of size l and contains the value of $-x$ (2-complement representation) on its l_x left-most bits, being the remaining $(l_y + 1)$ bits filled with zeros.
3. P is a bit vector of size l and it initially contains: on its l_x left-most bits the value 0, on the next l_y bits the value of y and in the final (right-most) bit the value 0. It represents a sum of partial products that at the end of the algorithm will contain the desired product of x and y.

The Booth multiplication process iterates the next steps l_y times, so at the end the bit vector P will contain the multiplication of x and y. At iteration i, the steps are:

1. If $P_0 = 0, P_1 = 0$, multiply the existing sum of partial products (P) by 2^{-1}. It is easy to see that this is an one place arithmetic shift to the right.
2. If $P_0 = 0, P_1 = 1$, add S to P and multiply by 2^{-1}.
3. If $P_0 = 1, P_1 = 0$, add A to P and multiply by 2^{-1}.
4. If $P_0 = 1, P_1 = 1$, multiply the sum of partial products by 2^{-1}.

Finally we drop the rightmost bit of P, so the product can be found on the l_x+l_y left-most bits of P.

According with this procedure, we have implemented a SAT encoding based on a circuit representation of it, where the basic building functions are full and half adders, plus some additional small circuits that control the right action to perform at each iteration of the above iterative process. The SAT encoding uses the sets of Boolean variables $\{A_i, S_i | 0 \leq i \leq l-1\}$ and $\{P_{i,j} | 0 \leq i \leq l_y+1, 0 \leq j \leq l-1\}$. The first set represents the bits of the vectors A and S and the second one represents the value of the vector P at the different iterations of the algorithm, so $P_{i,j}$ represents the value of bit j of P at iteration i. Iteration 0 refers to the initial value of P. Next, we present the clauses of the SAT encoding. For enforcing the values of the bit vectors A and S, we add unitary clauses that set the value of variables A_i and S_i as described before. Similarly, the initial value of P will be set on the variables $P_{0,j}$. The rest of clauses of the encoding simulate the different iterations of the algorithm, so we have a similar set of clauses for each iteration i of the algorithm. We use additional sets of variables for simulating the compu-

tations performed at iteration i: $\{T_i|1 \leq i \leq l_y\}$, $\{Q_{i,j}|1 \leq i \leq l_y, 0 \leq j \leq l-1\}$, $\{C_{i,j}|1 \leq i \leq l_y, 0 \leq j \leq l-1\}$. The set of clauses for iteration i is shown on Figure 4.

$$
\begin{array}{ll}
T_i \leftrightarrow P_{i,0} \oplus P_{i,1} & \\
Q_{i,j} \leftrightarrow T_i \wedge ((P_{i,0} \wedge A_j) \vee (P_{i,1} \wedge S_j)) & j = 0 \ldots l-1 \\
C_{i,0} \leftrightarrow Q_{i,0} \wedge P_{i,0} & \\
P_{i+1,j-1} \leftrightarrow Q_{i,j} \oplus P_{i,j} \oplus C_{i,j-1} & j = 1 \ldots l-1 \\
C_{i,j} \leftrightarrow ((Q_{i,j} \wedge P_{i,j}) \vee (Q_{i,j} \wedge C_{i,j-1}) \vee (P_{i,j} \wedge C_{i,j-1})) & j = 1 \ldots l-1 \\
P_{i,l-1} \leftrightarrow P_{i,l-2} &
\end{array}
$$

Figure 4. Boolean equations for iteration i of the Booth multiplication algorithm.

1.3. Pseudo Boolean Encoding

A linear *pseudo-Boolean constraint* (PB constraint) over Boolean variables is defined by $\sum_i c_i \cdot l_i \rhd p$ where c_i, the coefficients, and p, are integer constants, l_i are literals and $\rhd$ is one of the operators of $\{=, <, \leq, >, \geq\}$. Without loss of generality, these constraints can be rewritten to use the $\geq$ operator and positive coefficients (notice that $-c_i \cdot b_i$ can be rewritten as $c_i \cdot \neg b_i - c_i$). A coefficient c_i is said to be activated under a partial assignment if its corresponding literal l_i is assigned to true. Assuming that $\rhd$ is the $\geq$ operator, a pseudo-Boolean constraint is said to be satisfied under an assignment to its Boolean variables if the sum of its activated coefficients exceeds or is equal to k.

According to this we can encode the Factorisation problem as shown in Fig. 5.

$$
\begin{array}{ll}
\sum_{i=0}^{l-1} x_i 2^i \geq 2 & \\
\sum_{j=0}^{l-1} y_j 2^j \geq 2 & \\
\sum_{i=0}^{l-1} \sum_{j=0}^{l-1} z_{i,j} 2^{i+j} = k & \\
z_{i,j} - x_i - y_j \geq -1 & i,j = 0 \ldots l-1 \\
-2z_{i,j} + x_i + y_j \geq 0 & i,j = 0 \ldots l-1
\end{array}
$$

Figure 5. Pseudo-Boolean encoding for Factorisation

Firstly we try to avoid trivial solution and then we equal a binary representation of the number to factorize. Then we represent the partial products to perform factorization.

1.4. Quadratic Sieve

In order to compare our multipliers with other methods, one good benchmark is factorize large integers. To do so, we will use Quadratic Sieve method with the mathematical software SAGE to compare with our SAT-based multipliers. It is easy to see, that a SAT solver will find a solution to the factorization problem if we define its outputs.

Quadratic Sieve (QS) is known as one of the best methods to factorize integers, after Number Field Sieve. But it is still the best algorithm for integers up to 100 bits long.

The basics of QS are inspired on factorization Fermat's method, trying to find two numbers x and y such that $x \not\equiv \pm y \pmod{n}$ and $x^2 \equiv y^2 \pmod{n}$. This means that $(x-y)(x+y) \equiv 0 \mod n$, and we only need to check that $(x-y, x)$ is a non trivial division. As detailed in [11] there is at least a $1/2$ chance that the factor will be non trivial. The steps in doing so are the defined, firstly

$$Q(x) = (x + \lfloor \sqrt{n} \rfloor)^2 - n = \tilde{x}^2 - n,$$

and compute $Q(x_1), Q(x_2), \ldots, Q(x_k)$. From the evaluations of $Q(x)$, we want to pick a subset such that $Q(x_{i_1})Q(x_{i_2}) \ldots Q(x_{i_r})$ is a square, y^2. Then note that for all $x, Q(x) \equiv \tilde{x}^2 \mod n$. So what we have is that

$$Q(x_{i_1})Q(x_{i_2}) \ldots Q(x_{i_r}) \equiv (x_{i_1} x_{i_2} \ldots x_{i_r})^2 \mod n.$$

And if the conditions above hold, then we have factors for n.

2. Discrete Logarithm

Going back on elemental Algebra, we can define logarithm $log_a(h)$ operation as the solution to the equation $a^x = h$ over the real or complex numbers. Discrete Logarithm is analogue to the logarithm defined over the reals, but now defined over a finite cyclic group instead. So we can say that discrete logarithm is the solution x of the equation $a^x = h$ where a and h are elements of the finite cyclic group G.

Discrete Logarithm is a key concept for several cryptographic procedures such as the standard for digital signature DSS [6], that defines digital signature schemas based on Discrete Logarithm over integer modular arithmetic as well as over elliptic curves. For the first case, the standard specifies private key lengths –roughly speaking x– between 160 and 256 bits length.

All the known methods to tackle the Discrete Logarithm run in exponential time. The naivest approach is based on trial multiplication, encoded for SAT as explained in next subsection. Other methods performs faster, as Pollard's rho, detailed in subsection 2.2

2.1. Exponentiation

Suppose that you have two natural numbers a and x and you want to compute a^x. We can use x's binary representation, $x = \sum_{i=0}^{l-1} x_i 2^i$ to see that,

$$a^x = a^{2^0 x_0 + 2^1 x_1 + 2^2 x_2 + \ldots + 2^{l-1} x_{l-1}} = \prod_{i=0}^{l-1} a^{2^i x_i}.$$

Taking into account that x_i can only take binary values, we only have to multiply a^{2^i} elements whose x_i value is 1.

Making a sharper look into that technique, we can see that $a^{2^i} = (a^{2^{i-1}})^2$, which means that having a we can easily calculate next series's value squaring the previous one. We can see a modular scheme for this exponentiation method in Fig. 6 for an exponent l-bits long.

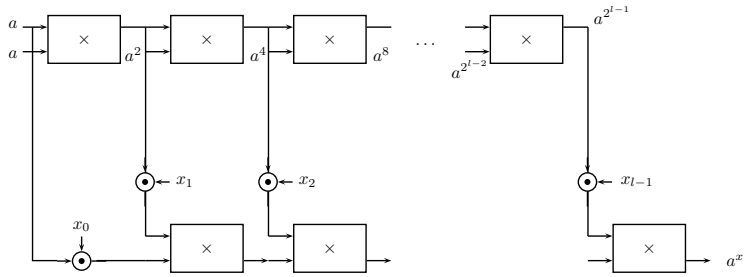

Figure 6. Modular design of an exponentiation circuit using multipliers and 1-bit exponentiators.

At this point in time, one thing left to say; an easy extension to $\mathbb{Z}_n$ of that fast exponentiation technique may be implemented using modulo n multipliers, with the advantage that any partial product will have length at most $2 \cdot \log n$, so the size of the exponentiation circuit for $\mathbb{Z}_n$ will remain polynomially bounded in the size of the input.

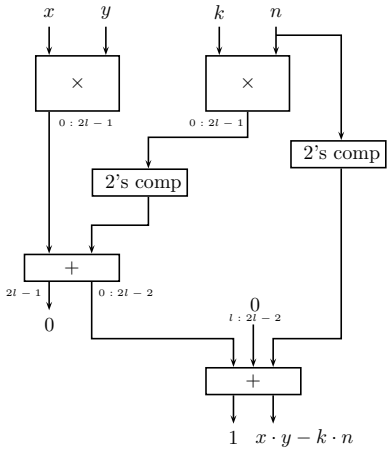

Figure 7. Modular design of a n-modulo multiplier.

In Fig. 7 there is a modular scheme for a n-modulo multiplier. Once again we use multipliers and adders explained before plus a two's complement module. What we want to encode is $x \cdot y = z \pmod{n}$, being x and y the multiplicands, n is the modulo, and the output $x \cdot y - k \cdot n$ for any $k \in \mathbb{Z}$. Two's-complement arithmetic is used to represent negative numbers. For ease of understanding we show Boolean equations for the two's complement module in Figure 8. We have A as an input vector, B an auxiliary vector containing the vector A negated, and finally C contains the output. All that vectors are l-bits long.

2.2. Pollard's rho

The Pollard's rho randomized algorithm for discrete logarithm [10], is based on finding a collision on a sequence of integers of the cyclic group $\mathbb{Z}_n$. Such a collision will be defined by two integers Y_i and Y_j of the iteration sequence $\{Y_0, Y_1, \ldots, Y_i, \ldots, Y_j\}$

$$\bigwedge_{i=0}^{l-1} B_i \leftrightarrow \bar{A}_i$$
$$\bigwedge_{i=2}^{l-1} (C_i \vee B_{i-1} \vee C_{i-1}) \wedge (\bar{C}_i \vee B_{i-1}) \wedge (\bar{C}_i \vee C_{i-1}) \wedge (C_i \vee \bar{B}_{i-1} \vee \bar{C}_{i-1})$$
$$(\bar{C}_1 \vee B_0 \vee C_0) \wedge (C_1 \vee \bar{B}_0 \vee \bar{C}_0) \wedge (C_1 \vee \bar{B}_0) \wedge (C_1 \vee \bar{C}_0)$$

Figure 8. Boolean equations of a Two's complement operator

generated with an iteration function $f : \mathbb{Z}_n \rightarrow \mathbb{Z}_n$ and such that $Y_i \equiv Y_j \ (mod\ n)$, where each Y_k of the sequence is of the form $Y_k = a^{k_1} h^{k_2}$. Then, once such collision is found, depending on the particular exponents of the two matching elements of the sequence, the solution to $a^x = h \ (mod\ n)$ may be obtained.

As with the quadratic sieve algorithm, we have used the implementation of this algorithm found in SAGE.

3. Experimentation and Results

In order to conduct our experimental investigation we have developed two generators of random problem instances for factorisation and discrete logarithm. For the factorization problem, we first search two primes of length $n/2$ and we took its product as the number to factorize. For the discrete logarithm, we first search a strong prime q using Gordon's algorithm [9], and then we search a generator g for the cyclic group and randomly select an element h from the group. So, we have that $g^x = h \pmod{q}$ is our discrete logarithm problem instance.

We have used two solving techniques for the experimental analysis:

- SAT solver: Precosat (v.236)[1]. Winner at the SAT Competition 2009 for the application category. We have used this algorithm for solving all the propositional encodings based on Boolean circuits for the problems considered.
- Pseudo-Boolean Solver: Minisat+ [4]. We have used this algorithm for solving the pseudo-Boolean encoding of the factorisation problems. It uses three main approaches to generate circuits in order to translate to CNF pseudo-Boolean linear problems, which then can be solved by a SAT-Solver. Those three approaches are:

  * Convert a pb-linear constraint to a BDD.
  * Convert a pb-linear constraint into a network of adders.
  * Convert a pb-linear constraint into a network of sorters.

  See [3] for more details about the above encodings for pb-linear constraints.

Our experiments have been run on machines with the following specifications: Rocks Cluster 5.2 Linux 2.6.18 Operating System, AMD Opteron 248 Processor clocked at 1.6 GHz, 1.0GB Memory, and GCC 4.1.2 Compiler.

Our SAT encoding for factorization (labeled as `Precosat Array` and `Precosat Booth`) are compared with PB-encoding (labeled as `Minisat+ Adders`, `Minisat+ BDD` and `Minisat+ Sorters`) are compared in Figure 9. As the reader may have advised, our SAT encodings for factorisation are defined taking any of the propositional encodings for integer multiplication and then fixing the output to the desired value n and

inserting additional constraints such that the factors x and y are not trivial. For illustration, these additional constraints are the first two constraints in the pseudo-Boolean encoding for factorisation of Figure 5. Figure 9 shows that our encoding using the Array multiplier is the best performing for factorization. Also we can see how the conversion of PB-encoding into a network of adders is performing quite well. That is because the approach is more or less the same, because both use adders. We will choose Array multiplier to compare SAT-encoding results with other benchmarks.

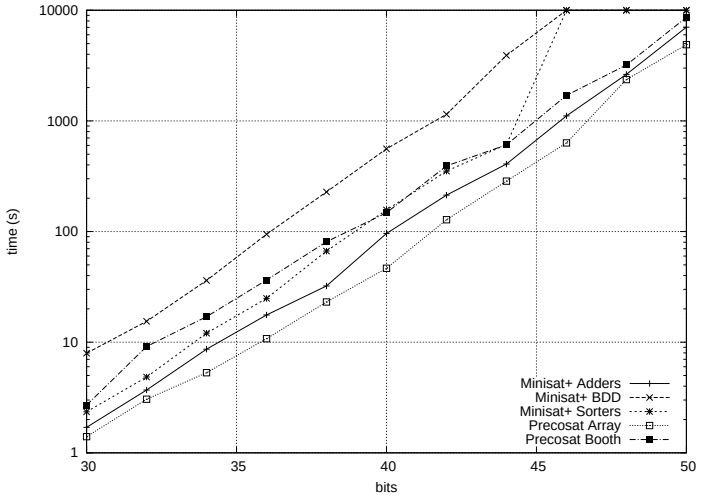

Figure 9. Results on Factorization

Figure 10 shows a comparison between the Array Multiplier SAT encoding and SAGE Quadratic Sieve implementation. We can see that our SAT approach performs significantly worse. The results show that with similar time bounds, Quadratic Sieve is able to factor integers of around 200 bits meanwhile the SAT encoding only factors integers of around 40 bits. The figure also shows a similar comparison between our SAT approach for solving the discrete logarithm and Pollard's Rho algorithm. This time, for a same time bound, the difference in the size of the discrete logarithm problems solved is about 4 times bigger for the specialized algorithm. To check the differences between our approach and the dedicated algorithm, we performed a linear regression in order to see the scaling cost of the specialized algorithms and the SAT-based approaches. We have a cost of $e^{0.4092 \cdot bits}$ for PrecoSAT factorization and $e^{0.06766 \cdot bits}$ for SAGE Quadratic Sieve. And $e^{0.8466 \cdot bits}$ for PrecoSAT discrete logarithm and $e^{0.3865 \cdot bits}$ for SAGE Pollard's Rho. So, these results indicate that there is still a lot of space for possible improvements of SAT encodings for these non-linear arithmetic problems.

4. Conclusions

We have presented SAT encodings of basic non-linear arithmetic operations: multiplication and modular exponentiation, used as the basic building blocks for encoding integer factorization and discrete logarithm as very challenging SAT instances. Our comparison of the performance of our SAT-based methods with the current best performing special-

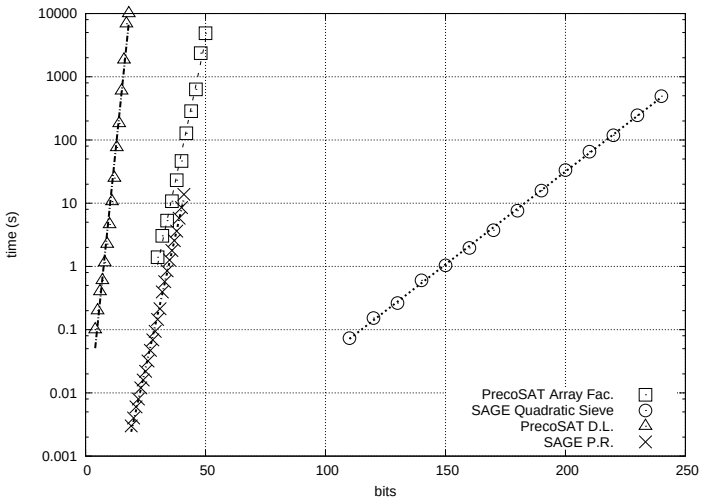

Figure 10. Results on Factorization and Discrete Logarithm

ized algorithms has shown that these problems are interesting challenges for the SAT research community, so that they deserve further study for trying to understand the limits on the performance of SAT encodings for such basic problems.

In the future, we expect to use our SAT encodings for encoding Elliptic Curves problems, where the lack of specialized algorithms for them may make worth the study of the performance of SAT-based algorithms.

References

[1] Armin Biere. Precosat version 236. http://fmv.jku.at/precosat.

[2] Andrew D. Booth. Signed binary multiplication technique. *Q J Mechanics Appl Math*, 4(2):236–240, 1951.

[3] Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into SAT. *JSAT*, 2(1-4):1–26, 2006.

[4] Niklas EÃl'n and Niklas SÃűrensson. Translating pseudo-boolean constraints into sat. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26, 2006.

[5] Claudia Fiorini, Enrico Martinelli, and Fabio Massacci. How to fake an rsa signature by encoding modular root finding as a sat problem. *DISCRETE APPL. MATH*, 130:101–127, 2003.

[6] Patrick Gallagher, Deputy Director Foreword, and Cita Furlani Director. Fips pub 186-3 federal information processing standards publication digital signature standard (dss), 2009.

[7] Henry A. Kautz, David A. McAllester, and Bart Selman. Encoding plans in propositional logic. In *KR'96*, pages 374–384, 1996.

[8] Henry A. Kautz and Bart Selman. Unifying sat-based and graph-based planning. In *IJCAI'99*, pages 318–325, 1999.

[9] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1996.

[10] J.M. Pollard. Monte carlo methods for index computation mod p. *Mathematics of Computation*, 32:918–924, 1978.

[11] C Pomerance. The quadratic sieve factoring algorithm. In *Proc. of the EUROCRYPT 84 workshop on Advances in cryptology: theory and application of cryptographic techniques*, pages 169–182, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[12] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.