Satisfy This: An Attempt at Solving Prime Factorization using Satisfiability Solvers

Stefan Schoenmackers, Anna Cavender
Department of Computer Science
University of Washington
Seattle, WA 98195
stef,cavender@cs.washington.edu

Abstract

Prime factorization of large integers is a difficult problem in number theory [1,6]. Even the best known methods of factoring an integer composed of two primes would, given a reasonably large integer, take longer than the lifetime of the universe. It is because of this difficulty that RSA is believed to be secure (for large enough keys). This paper discusses the authors' attempt at solving the prime factorization problem using satisfiability solvers and well as previous work on this problem. The authors' solution is then analyzed empirically in comparison with two other solutions.

Introduction

Problem Definition

For this paper, we are concerned with a small, yet important, subset of the prime factorization problem: determining the factors of a large integer known to be composed of only two primes of equal bit-length. Namely, given an integer N which we know is of the form $N = p * q$ where p and q are prime, and $|p| = |q|$, how quickly can a satisfiability solver determine the actual values p and q?

Motivation

The motivation for doing this is three-fold. First, this problem is one of the outstanding challenges for satisfiability solvers [1]. Many instances of "hard" satisfiability problems can be created by posing the integer factorization problem as a satisfiability problem. Intuitively, this problem is difficult since there are only two solutions in the entire input space (namely, $p * q$ and $q * p$), thus the odds of finding a solution decrease exponentially in the size of the inputs. Furthermore, half of the bits in the result N depend directly on all of the bits of p and q, so determining the bits of p and q requires tracing and propogating complex effects through many clauses.

Secondly, the authors found only a small amount of previous work on this problem, indicating that it probably hasn't recieved as much research attention as it should.

Finally, this is a problem of practical importance. Most widely used cryptosystems that keep web traffic, remote access, and sensitive files secure use the RSA public-key cryptosystem at some point in their procedure. The RSA cryptosystem is based on the difficulty of factoring large composites of two primes (i.e. numbers of the form this paper examines.) Advances in number theory have greatly increased the speed of factoring, but it is still believed to be impractical for larger keys. If modern satisfiability solvers can decrease the required time, then it would indicate current cryptosystems aren't as secure as they are believed to be, and should either increase their keylength, or switch to some other method such as elliptical-curve cryptography.

Previous results have been much slower than brute-force search, and impractical for input sizes beyond about 50 bits. Due to these results, most people believe that satisfiability solvers are impractical to use for this problem, unless the underlying structure of the problem can be somehow captured and efficiently represented. This work represents the the authors' attempt to exploit the local structure inherent in multiplication.

Layout

We begin by defining concepts that will be used throughout the paper. We then discuss previous work in the area. After discussing our implementation, we show the results of our emperical comparision to the previous work. Finally we conclude the paper and examine potential future research as extentions of this initial study.

Definitions

In this section, we briefly introduce definitions that will be used throughout the remainder of the paper.

Satisfiability

A Boolean variable α is a symbol that can take on the value $true$ or $false$. Boolean operations AND($\wedge$), OR($\vee$), and NOT($\neg$) are used to combine these variable into Boolean sentences (ϕ). ϕ is satisfiable (SAT) if and only if there exist an assignment for each variable that causes ϕ to evaluate to $true$. Deciding if a given ϕ has a satisfying assignment is considered the quintessential NP-complete problem.

Conjuctive Normal Form

Any Boolean sentence can be converted into Conjuctive Normal Form (CNF). A sentence is in CNF if it a conjunction of disjunctions. In other words, the sentence can be separated into clauses consisting entirely of variables connected by OR operations and each clause in connected by an AND operation. For example, $(\alpha \vee \beta) \wedge (\gamma \vee \neg\beta)$ is in CNF while $((\alpha \wedge \beta) \vee \gamma)$ is not.

Hardware Adders and Multipliers

Addition and multiplication are two of the most basic operations performed at the low-level hardware within a computer's processor by circuits of AND and OR gates. The basic block of binary addition is the Full Adder in Figure 1 where a and b are the ith bit in the binary representation of the two numbers being added, cin is the carry bit from the previous position, sum is the result of adding those two bits, and $cout$ will be the carry to the next Full Adder. So, adding two n-bit numbers will require a chain of n Full Adders. While much computer architecture research has been devoted to improving the speed of addition, the fundamental process still reduces to a series of Full Adders.

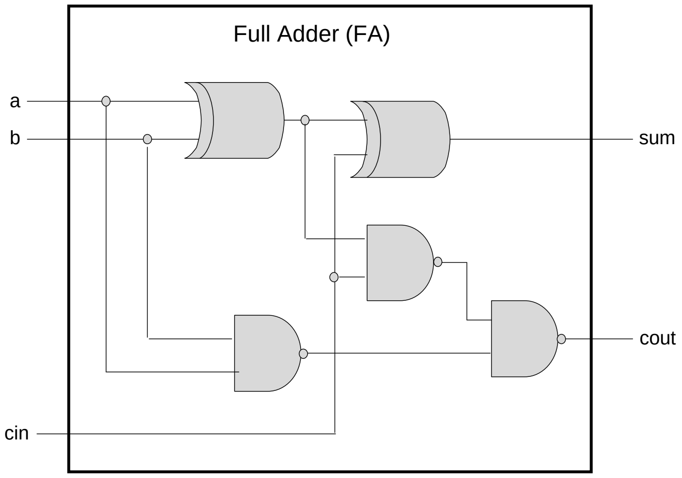

Figure 1: Full Adder. Computes the sum and carry out of the bits a and b taking into account the carry in from the previous bits.

The process of multiplication is simply a series of shifts and additions. There are several procedures for doing this, each with varying tradeoffs in terms of complexity and circuit speed. For this paper, we focus on two different forms of Carry-Save Multiplication, discussed in the Section labeled Multiplier.

Previous Work

Previous work in this area has concentrated on the factorization of *any* number. Thus, if the Boolean sentence representing the input number N is *un*satisfiable then N is prime. Conversely, if the sentence is satisfiable then N is obviously not prime, but its factors may be any natural number. The problem can be greatly reduced if we narrow our focus to numbers that have only two factors p and q where both p and q are prime. In fact, we can condense the problem further by requiring that p and q have equal length in their binary representations. These two constraints make the problem more manageable by eliminating a large section of the input space, since we need not consider more or less than $|N|/2$ bits per input.

The general idea of factoring using satisfiability is to simulate the circuits of a hardware adder and multiplier on the bits of potential p's and q's, while setting the output bits the same as the desired N. The result is a Boolean sentence such that each variable corresponds to either one of the bits in p or q or values for the circuit wires. This Boolean sentence can then be concerted into CNF (Conjuctive Normal Form) so that a satisfiability (SAT) solver can solve it. If the Boolean sentence can be solved (i.e. there exists an assignment for each variable such that the sentence is true), then each bit in p and q will have the value of its corresponding Boolean variable and we have found the p and q that compose N.

The CNF Generator by Paul Purdom and Amr Sabry implements both a carry-save multiplier and a Wallace-tree multiplier, giving the user this option. Additionally, they allow the user to simulate either an n-bit or a fast adder, which yield the same result, but differ in the structure of how the additions take place. Their circuit simulator requires that the first factor have at most $|N| - 1$ bits and the second have at least $|N|/2$ bits (rounded up when N is odd). Their program can be used online at http://www.cs.indiana.edu/cgi-pub/sabry/cnf.html. They also generously provide the source code, so that it may be run on a local machine.

The FactoringAsSat project by Henry Kautz and Shane J. Neph similarly produces a sentence that is unsatisfiable if the input is prime, or has a satisfying assignment representing two of the divisors of a non-prime input. Unlike Purdom and Sabry's generator, FactoringAsSat assumes that each factor may be up to $|N|$ bits long, and they add additional constraints that prohibit either factor from being 1, since $1 * N$ is always a solution, but is not the desired one.

Both of these problem generators do a low-level simulation of the multiplier circuit. They save the outputs of each adder and/or half-adder, which is conceptually easy to follow and construct, but requires a large number of intermediate variables to store all these temporary values.

Proposed Improvements

Unlike previous work that saves all the intermediate values, we propose tracing the full circuit, and only retaining variables for the bits of p and q. Instead of storing each of these temporary values as a new variable, we construct a boolean expression by tying the outputs of one adder to the inputs of the next. This will result in a complex boolean formula, which we then convert to CNF, and find its solution using a satisfiability solver.

The motivating idea is that the runtime of a satisfiability problem is typically dependent on the number of variables it must assign, so decreasing the number of variables should decrease the time it takes to factor. This process will result in more clauses, but we hypothesize that the advanced techniques in modern satisfiability solvers, such as watched clauses and clause learning, will allow us to overcome the increase in the number of clauses.

The Multiplier

For our project, we simulated a multiplier circuit and set its output bits to correspond to N. We then use this multiplier to construct a Boolean sentence that, if satisfiable, produces the two prime factors of equal binary length of a large integer input. After transforming that sentence into CNF, a satisfiability solver generates an assignment for the two factors.

Fundamentally, multiplication is just a series of shifts and additions. For example, to multiply $b = b_3b_2b_1b_0$ by $a = a_2a_1a_0$, we start with the rightmost bit of one of the terms. If that bit (a_0) is 1, we add one copy of the other term (b) to the final result, if that bit is 0, we add no copies of that term (i.e. add 0). For each successive bit a_i in a, we shift b left by i, and add it to the final result if a_i is 1. The circuit calculates this by adding $b \wedge a_0 + (b << 1) \wedge a_1 + (b << 2) \wedge a_2 + \ldots$ where $b << i$ indicates a left shift of b by i bits.

There were two choices to be made for our simulation circuit. Namely, which type of adders we would simulate, and how we would chain them to get the final product. We used a simple ripple-adder for each of our adders, where the carry-out from each Full Adder is tied to the carry-in of the next. This is far from state of the art in computer architecture, but all of the improvements that speed up adder hardware add complexity to the circuit. Since we want to keep the number of variables and clauses small, we chose this simple ripple-adder so that we wouldn't have to simulate the additional complexity of other methods.

The second choice we made was how to perform the sequence of additions. The obvious choice is to keep a running sum to which we add each successive term. An alternative is to recursively add adjacent terms, thus building the sequence of additions as a tree. The leaves of the tree are simply the shifted and ANDed terms ($(b << i) \wedge a_i$), and each internal node just sums two adjacent nodes. The final result is the output of the adder at the top of the tree. Both forms require the same number of additions, they merely differ in the order of those additions. Some preliminary experiments showed that the final CNF generated using this tree structure had far fewer clauses than one using the running sum, so we used this tree-multiplier for our experiments. We believe that there is a fair amount of related material in adjacent terms for each adder, which can be exploited to make the final CNF simpler.

The Conversion to CNF

Once the multiplier produces a Boolean sentence ϕ, we normalize this sentence to CNF where every clauses consists of ORs ($\vee$) of variables that are ANDed ($\wedge$) together. To do this we implimented the following DeMorgan Laws in three steps: moving negations inward, distributing nested operations, and flattening of clauses. The first step involves moving all negations ($\neg$) inward so that only the variables (as opposed to entire clauses) are negated. For example:

$$\neg(\neg\alpha) \equiv \alpha$$
$$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta)$$
$$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta).$$

If either α or β are clauses, the negation will recursively propogate inward until a literal is found. The second step distributes all of the nested operations. For example:

$$(\alpha \vee (\beta \wedge \gamma)) \equiv (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$$

Again, if α, β, or γ are clauses, the program recursively applies this rule until all clauses are in CNF. Finally, in order to detect and eliminate duplications (such as $\alpha \vee \alpha \equiv \alpha$) and contradictions (such as $\alpha \vee \alpha \equiv true$) we find nested clauses composed solely of ORs and flatten them into one clause.

Modifications to the Original Design

Our ideal going into this project was to construct this multiplier using as few variables as possible: only the bits of the input. We wanted to construct a complex boolean formula that calculated the output using only the input variables, convert it to CNF, and then use a satisfiability solver. We quickly realized that this would not be feasible. The clauses became deeply nested, and the conversion to CNF took far too long for even small composites such as 35. Furthermore, the conversion process generated an exceedingly large number of clauses, such that using this method directly became intractable.

We overcame this obstacle by making a few minor adjustments. First of all, we noticed that the cascade of carry-bits was contributing to a large part of the complexity, so instead of letting them propagate, we added a new variable that corresponded to the value of each. This only increased the number of variables by a moderate amount, and it greatly reduced the number of clauses. However, this was still intractable for larger inputs. To get around this issue, we used two different methods to periodically save the output bits from each of the adders.

Alternating-Save

One of the methods we tried saved the adder output bits at every-other level in the addition tree, so we designate it as Alternating-Save. This still allows for a fair amount of local-similarity that we could exploit (since there are similarities in adjacent nodes of the addition tree), but greatly reduced the time to create the CNF. The output from this method had fewer variables than all but the Left-Save method (described next), and generated fewer clauses than any of the other methods. So while this falls short of our ideal of minimizing the number of variables, it captures a similar essence while being tractable.

Left-Save

The other method we tried saved the adder outputs for the left adder at each node in the tree. The idea behind this is to retain some of the complex nested clauses at each addition (and allow deeper similarities to be exploited), while saving enough of them to still make the problem tractable. This method used the smallest number of variables of any cnf generators we tried, but it also created 3-4 times more clauses than any other method we tried, and took an order of magnitude longer to run than any of the other methods. It is a good comparison for the tradeoffs between number of clauses and number of variables.

Experiments

We empirically tested our program with the CNF Generator [4] and FactoringAsSat [5] against both zChaff [2] and Walksat [3]. We examined these three generators using 45 composites, whose prime factors varied from 12 to 20 bits each. In our initial experiments we noticed a large variance in runtime when factoring different numbers. Since we weren't able to determine the cause of this discrepancy, we attributed it to the random variations in the choice of assignments. To compensate for potential inconsistancies, we examined the average runtime over 5 different numbers per bit-length.

For each test case, we generated an approriate CNF using all three generators. We examined both the left-save and the alternate-save variations of our generator. For the CNF Generator, we tested both carry-save and wallace multipliers using the fast-adder option. Since we knew a priori that each of the factors has $|N|/2$ bits, we added clauses to each CNF that would force any higher bits to be zero. This ensures that we don't penalize other generators by providing only ours with this additional information.

Walksat performs well in instances where the solution-space is dense, thus it wasn't well suited for the sparcity of this problem. Becuase, it was unable to correctly factor any of the 45 numbers in our test set, we omit it from further disucssion and focus solely on zChaff.

Due to time constraints, we also restricted the runtime of zChaff by only allowing it to run for 1 hour for each test case. Only a few test cases were unable to complete within the hour and, as discussed below, all were with the problematic Wallace-Multiplier.

Unsimplified CNF

For the first part of our experiment we provided zChaff with the default output of each generator. The only modification was the additional clauses setting the higher bits to zero. As expected, our Left-Save generator resulted in considerably more clauses (see Figure 2) than other generators with the tradeoff of considerably fewer variables (see Figure 3).

Also as expected, the Alternate-Save case had relatively few variables, but surprisingly it also appeared to have the fewest number of clauses. This is mainly due to extra clauses in the other generators that account for bits in the factor larger than n. These unneeded clauses are eliminated by a SAT-simplifier as discussed in the next section.

All methods performed about the same for smaller inputs, but on larger inputs, the Alternate-Save method found a solution 20% faster on average than any other method, with the next best being FactoringAsSat (see Figure 4). Both outputs from the CNF Generator performed significantly worse on these larger inputs. The wallace-multiplier in particular incorrectly generated some of the CNF descriptions for cases of length 19 and all descriptions for length 20 bits, and was

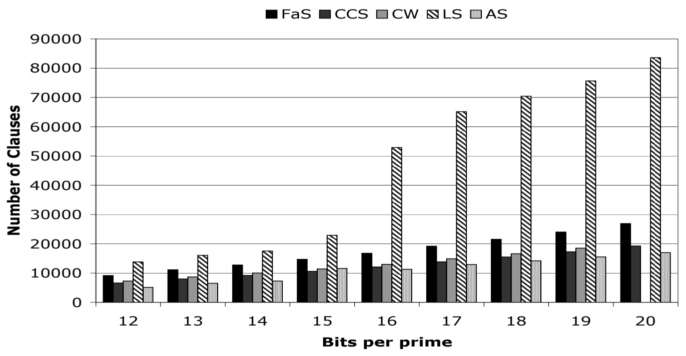

Figure 2: Clause counts for the *unsimplified* output each of the five generators when tested on primes of varying bit-lengths. FaS = FactoringAsSat, CCS = CNF Generator with Carry-Save, CW = CNF Generator with wallace, LS = Left-Save, AS = Alternate-Save.

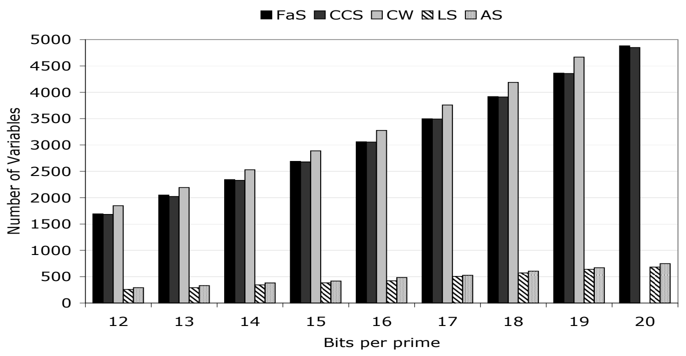

Figure 3: Variable counts for the *unsimplified* output each of the five generators when tested on primes of varying bit-lengths.

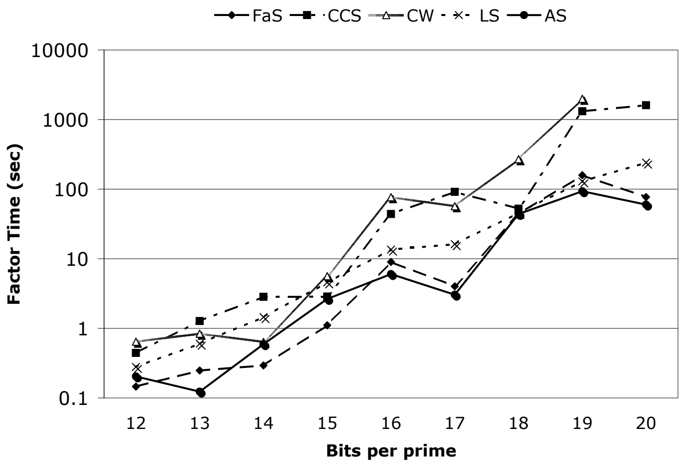

Figure 4: The results of running zChaff with unsimplfiied outputs of five generators.

therefore unable to solve them. The CNF Generator when run with the carry-save adder was able to correctly factor all test cases, but it approached the 1 hour time limit for many

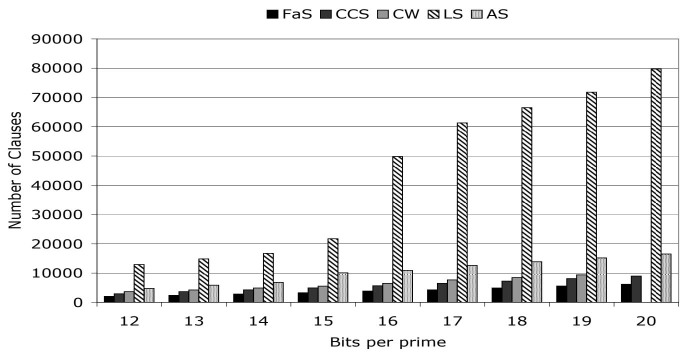

Figure 5: Clause counts for the *simplified* output each of the five generators when tested on primes of varying bit-lengths. FaS = FactoringAsSat, CCS = CNF Generator with Carry-Save, CW = CNF Generator with wallace, LS = Left-Save, AS = Alternate-Save.

of the larger ones.

Simplified CNF

For the second part of our experiment, we ran the CNF formulas through the Hypresource CNF Simplifier [7] before feeding them to the satisfiability solver. According to the authors, this simplifier "does a rather remarkable job of simplifying formulas, working best on structured formulas." Since this problem is highly structured (a cascade of adders), the simplifier should reduce the input to the SAT-solve, and ideally allow it to run more efficiently.

Surprisingly, this was not the case. It made the best performing generators (Alternate-Save and FactoringAsSat) worse by a factor of 4-5, and it made the worst performing generators (Left-Save and CNF Generator) better by a factor of 2 and .5 respectively. This is an interesting phenomenon that may be due to the elimination of some structure inherent in the unsimplified representations, but is a good opportunity for further research.

As expected, the simplifier reduces the number of clauses and variables for all test cases. The simplifier eliminates the extraneous clauses that represent the extra bits of the factors which are known to be false. As expected, with these clauses removed, the other methods have fewer clauses, but more variables than our methods. Again, Figures 5 and 6 visually show the tradeoff of clauses to variables using our method.

Conclusions

Our method of reducing the number of variables by saving adder outputs at every-other level in the tree generated fewer variables, and was able to factor the numbers 20% faster than the other methods on average. The Left-Save method was an interesting method that used the fewest variables, but its significantly larger clause count contributed to its overall worse performance. The FactoringAsSat method was highly competitive despite its generalizations (i.e. assuming factors can be any length), and in some test cases performed better than the Alternate-Save method.

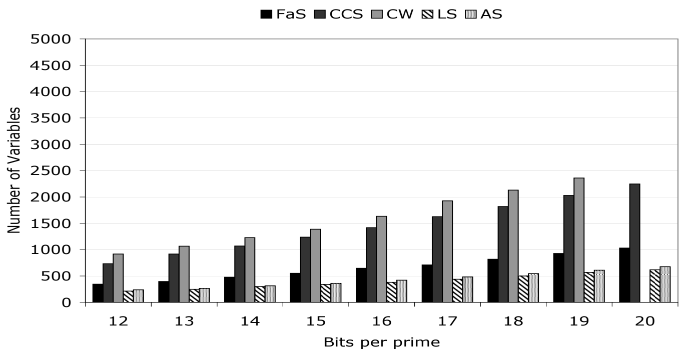

Figure 6: Variable counts for the *simplified* output each of the five generators when tested on primes of varying bit-lengths.

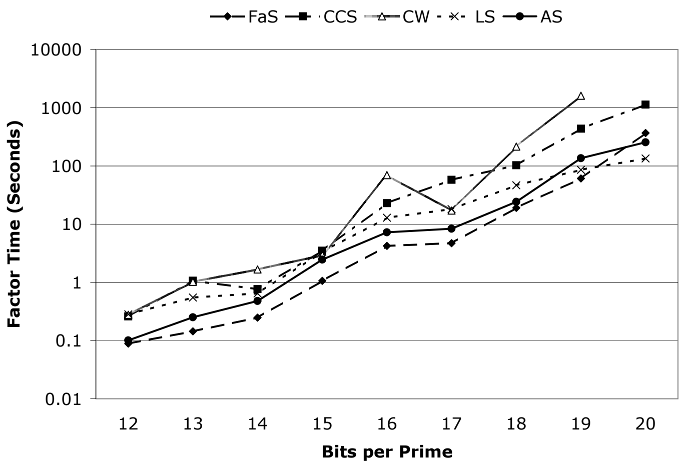

Figure 7: The results of running zChaff with simplfiied outputs of five generators.

While this was interesting work, for the numbers we used it is still much slower than brute force (takes only a few seconds to find both 20-bit primes in all of the largest examples.) It would be interesting to see how this method compares to brute-force and to more intelligent factoring algorithms such as the General Number Field Sieve, the currently fastest known factoring algorithm, as the sizes of the primes scale up.

Research Directions

This project left us with many ideas for future work. We found ourselves wishing for more time and more resources on several occasions. It would be very interesting to create or modify an existing SAT solver to take advantage of this particular problem domain. For example, it may be useful to encode the SAT solver with information about number theory, boolean logic, and DeMorgan's Laws. The new SAT solver could also take advantage of the structure of clauses that are likely to occur using a specific multiplier.

In this project we implemented the Carry-Save Multiplier, but it may be the case that a Carry-Select, Carry-Skip, or

Carry-LookAhead would be more efficient for this specific problem. A comparision of all four may provide insight into other improvements to either the SAT solver, the simplifier, or the CNF conversion.

Obviously, it would be a valuable endeavor to investigate the precise functionality of the Hypresource simplifier. Specifically, we've noticed that its output occationally contains clauses that have equivalent meaning, but in which the variables are ordered differently (such as $(a \vee b \vee c)$ and $(b \vee c \vee a)$). Thus it seems plausible to sort the variables in each clause first, then sort the clauses, and then look for duplicates.

Finally, it would have been fun to run some very very big numbers, possibly on multiple, high-performance compters, to see how our creation measures up to known records.

Reference List

1. Stephen A. Cook and David G. Mitchell, Finding Hard Instances of the Satisfiability Problem: A Survey. In Satisfiability Problem: Theory and Applications. Du, Gu and Pardalos (Eds). *Dimacs Series in Discrete Mathamatics and Theoretical Computer Science, Volume 35*

2. Zhaohui F.(2004), zChaff version 2004.5.13 Retrieved October 26, 2004 from, http://www.princeton.edu/ chaff/zchaff.html

3. Kautz, H., Selman, B.(2004), Walksat version 45(Released February 26, 2004). Retrieved October 27, 2004 from, http://www.cs.washington.edu/homes/kautz/walksat/

4. Purdom, P., Smabry, A.(2004), CNF Generator for Factoring Problems (June 5, 2004). Retrieved October 22, 2004 from http://www.cs.indiana.edu/cgi-pub/sabry/cnf.html

5. Kautz, H., Neph, S.J.(2004), FactoringAsSat Received from the authors October 22, 2004.

6. Srebrny M.(2004), Factorization with SAT classical propositional calculus as a programming environment

7. Bacchus, F., Winter , J. (2003) Effective Preprocessing with Hyper-Resolution and Equality Reduction, *Sixth International Symposium on Theory and Applications of Satisfiability Testing*

Appendix A (Work Distribution and Acknowledgements) The first author (Stef) created the multiplier and the second author (Anna) implemented the conversion to CNF. Code for the FactoringAsSat [5] was generously provided by Henry Kautz for use in our empirical studies. The CNF Generator [4] was also used for empirical studies. The two SAT solvers used for the experiments were zChaff [2] and Walksat [3]. Code for the simplifying procedure called Hypresource [7] was used unmodified.