UNIVERSITY OF OSLO
Department of Informatics

Master thesis

SAT-based preimage attacks on SHA-1

Vegard Nossum

November, 2012

Abstract

Hash functions are important cryptographic primitives which map arbitrarily long messages to fixed-length message digests in such a way that: (1) it is easy to compute the message digest given a message, while (2) inverting the hashing process (e.g. finding a message that maps to a specific message digest) is hard. One attack against a hash function is an algorithm that nevertheless manages to invert the hashing process. Hash functions are used in e.g. authentication, digital signatures, and key exchange. A popular hash function used in many practical application scenarios is the Secure Hash Algorithm (SHA-1).

In this thesis we investigate the current state of the art in carrying out preimage attacks against SHA-1 using SAT solvers, and we attempt to find out if there is any room for improvement in either the encoding or the solving processes.

We run a series of experiments using SAT solvers on encodings of reduced-difficulty versions of SHA-1. Each experiment tests one aspect of the encoding or solving process, such as e.g. determining whether there exists an optimal restart interval or determining which branching heuristic leads to the best average solving time. An important part of our work is to use statistically sound methods, i.e. hypothesis tests which take sample size and variation into account.

Our most important result is a new encoding of 32-bit modular addition which significantly reduces the time it takes the SAT solver to find a solution compared to previously known encodings. Other results include the fact that reducing the absolute size of the search space by fixing bits of the message up to a certain point actually results in an instance that is *harder* for the SAT solver to solve. We have also identified some slight improvements to the parameters used by the heuristics of the solver **MiniSat**; for example, contrary to assertions made in the literature, we find that using longer restart intervals improves the running time of the solver.

Acknowledgements

First and foremost, I thank my thesis advisors, **Fabio Massacci** and **Martin Giese**, for their insights, help, and feedback, and for directing me back onto the right path when I lost sight of my goal. Secondly, I thank **Mate Soos** for participating in countless discussions about SAT solving. Thirdly, I thank my wife, **Cristina Onete**, my parents, **Erik** and **Tove Nossum**, and my family-in-law, for their love and enduring support. Thank you.

Contents

List of figures

List of tables

Introduction

"Our greatest weakness lies in giving up. The most certain way to succeed is always to try just one more time."

Thomas A. Edison

Hash functions are fascinating — given the specification of a hash function $\mathbf{f}$, we can easily calculate the hash value $\mathbf{h} = \mathbf{f}(\mathbf{m})$ of any message $\mathbf{m}$. However, finding a message $\mathbf{m}$ (called a *preimage*) so that it hashes to a specific value $\mathbf{h}$ is extremely difficult. We know exactly what $\mathbf{f}$ does, so *why* is it so difficult? Even worse, we know that every message has practically an infinite number of preimages, so how difficult could it be?

It is not difficult to construct an algorithm which finds preimages: using brute force, we can simply calculate the hash value of every possible messages in turn until we find one that hashes to the right value. However, this procedure would on average take longer than the lifetime of the universe to complete. Thus, the challenge lies in *efficiently* finding such a message.

SAT solvers are programs which essentially carry out a brute force search. The difference between a SAT solver and a naïve brute force search is that the SAT solver can exploit hidden algebraic structures in the problem it is trying to solve. Put simply, in the context of searching for preimages, the SAT solver can deduce that whole classes of messages (messages with a certain structure) can never be preimages — and therefore skip them during the search.

This thesis focuses on preimage attacks on a specific hash function, SHA-1, using SAT solvers. We remark that the goal of the thesis is *not* to produce a fully-fledged attack; rather, the goal is to assess the current state of the art and find out where there is room for improvement. A large part of the thesis is dedicated to the individual experiments where we assess the impact of different encodings and SAT solving techniques on the running time of the solver.

The use of SAT solvers to speed up brute force attacks against cryptographic primitives was for the first time thoroughly investigated in [Massacci and Marraro, 2000]. Their paper describes an attack on reduced-round versions of the block cipher DES. Their results were not particularly encouraging; however, SAT solvers have successfully been put to use in collision attacks against hash functions (e.g. [Mironov and Zhang, 2006]) and attacks against stream ciphers (e.g. [Courtois et al., 2008]).

The thesis is structured as follows: In chapter 1, we give a more formal introduction to hash functions, SHA-1, and SAT solving in general. In section 1.5 we give a short introduction to algebraic and logical cryptanalysis (i.e. the use of SAT solvers to carry out attacks against cryptographic primitives) and review the published literature in this field.

In chapter 2, we discuss how to encode SHA-1 and preimage attacks as a SAT problem, including encodings used in the literature and our proposed modifications. We also briefly describe our implementation of the instance generator which is used in most of our experiments.

In chapter 3, we discuss the different statistical methods that we need in order to ensure that the results of our experiments are accurate and trustworthy (e.g. not simply due to chance variation).

In chapter 4 we carry out the experiments themselves. First, we test the performance of 12 freely available SAT solvers in order to identify the best unmodified solver(s) for our type of problems. Then we proceed to examine the effect of various difficulty parameters — e.g. how does the number of rounds in the hash function affect the running time of the SAT solver? The following experiments deal with different encodings, different constraint types, and different SAT solver heuristics (including e.g. branching heuristics and restart heuristics).

We summarise our findings and suggest a few topics for further research in chapter 5.

For Cristina, who opened my eyes...

Chapter 1

Background

1.1 Cryptographic hash functions

Cryptographic hash functions are functions designed specifically so as to easily to compute a fixed-length message digest ("hash") given a message with arbitrary length, while making it difficult to gain any information about a message given its hash (i.e. by obscuring the relationship between messages and their hashes). In particular, a cryptographic hash function $\mathbf{f}$ should have the following properties:

- **Preimage resistance.** Given a hash $\mathbf{H}$, it is infeasible to find a message $\mathbf{M}$ such that $\mathbf{f}(\mathbf{M}) = \mathbf{H}$.

- **Second-preimage resistance.** Given a message $\mathbf{M}$, it is infeasible to find another message $\mathbf{M}'$ such that $\mathbf{f}(\mathbf{M}') = \mathbf{f}(\mathbf{M})$.

- **Collision resistance.** It is infeasible to find distinct messages $\mathbf{M}$ and $\mathbf{M}'$ such that $\mathbf{f}(\mathbf{M}) = \mathbf{f}(\mathbf{M}')$.

The relationships between these properties are as follows: if a hash function is *not* preimage resistance, then it is also *not* collision and second-preimage resistant. Furthermore, if a hash function is *not* second-preimage resistant, then it is also *not* collision resistant.

An attack on a cryptographic hash function generally refers to any algorithm that can find what the above properties specify is infeasible to find, regardless of whether it is efficient (e.g. we still call a brute force algorithm an attack, even though it is infeasible in practice). Since these attacks generally work by hashing messages and verifying that the hashes match, the difficulty of an attack is usually given in the number of messages that must be hashed before a correct solution is found. A generic attack on preimage resistance is a brute force search which tries all messages until one is found that hashes to the given value; since each bit in the hash has a probability of $1/2$ of being correct (by chance), we expect to find a correct solution after trying approximately 2^n messages, where n is the number of bits in the hash.

A generic attack on collision resistance is the birthday attack [Katz and Lindell, 2007]. By a probabilistic argument, one can show that no more than $2^{n/2}$ messages must be hashed on average to find a collision, where n is the number of bits in the hash, a constant which depends on the hash function in question.

1.2 SHA-1

SHA-1 is a cryptographic hash function that was designed by the National Security Agency (NSA) and published in 1995 [SHA, 1995] as a follow-up to its flawed predecessor SHA-0 from 1993. It has a fixed hash size of 160 bits and is based on the Merkle-Damgård construction [Katz and Lindell, 2007]. The Merkle-Damgård construction is a way to construct a collision-resistant hash function of arbitrary input size from a collision-resistant compression function with a fixed input size (see figure Figure 1.1).

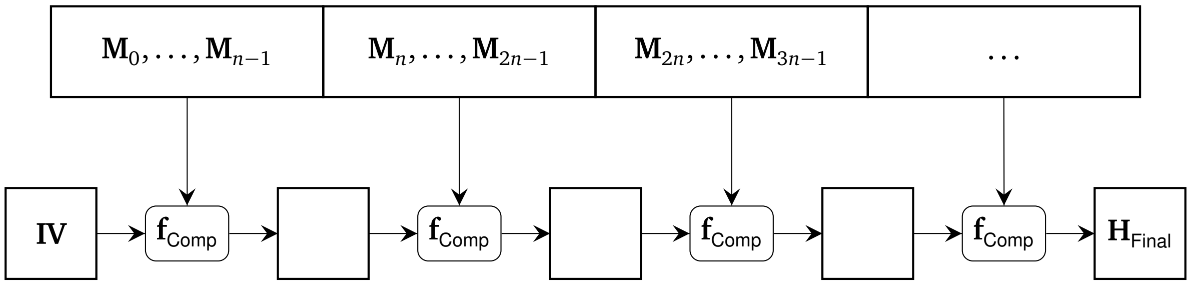

$\mathbf{M}_0, \ldots, \mathbf{M}_{n-1}$
$\mathbf{M}_n, \ldots, \mathbf{M}_{2n-1}$
$\mathbf{M}_{2n}, \ldots, \mathbf{M}_{3n-1}$
$\ldots$
IV
$\mathbf{f}_{\text{Comp}}$
$\mathbf{f}_{\text{Comp}}$
$\mathbf{f}_{\text{Comp}}$
$\mathbf{f}_{\text{Comp}}$
$\mathbf{H}_{\text{Final}}$

Figure 1.1: The Merkle-Damgård construction.

SHA-1 has a compression function that takes a 160-bit hash value and a 512-bit part of the message and outputs another 160-bit hash value. The first hash value is called the initialisation vector and is a constant. The last hash value is used as the final output of the hash function.

Since an attack against the compression function implies an attack against the hash function, we will in this thesis only focus on attacks against the compression function. The SHA-1 compression function $\mathbf{f}_{\text{Comp}}$ is given as follows:

$$\mathbf{f}_{\text{Comp}}(\mathbf{H}_0 \| \cdots \| \mathbf{H}_4, \mathbf{M}_0 \| \cdots \| \mathbf{M}_{15}) = (\mathbf{H}_0 + \mathbf{a}_{80}) \| \cdots \| (\mathbf{H}_4 + \mathbf{a}_{84}), \tag{1.1}$$

where $\|$ denotes concatenation of sequences, $\mathbf{H}_0, \ldots, \mathbf{H}_4$ are sequences of 32 bits (the initialisation vector), $\mathbf{M}_0, \ldots, \mathbf{M}_{15}$ are sequences of 32 bits (the part of a message), and $\mathbf{a}_{80}, \ldots, \mathbf{a}_{84}$ are sequences of 32 bits. The intermediate states $\mathbf{a}_t$ are given as follows:

$$\mathbf{a}_t = \begin{cases} \mathbf{H}_t, & 0 \le t \le 4 \\ \mathbf{a}_{t-1} + \mathbf{f}_t(\mathbf{a}_{t-2}, \mathbf{a}_{t-3}, \mathbf{a}_{t-4}) + \mathbf{a}_{t-5} + \mathbf{k}_t + \mathbf{W}_{t-5}, & 5 \le t \le 84 \end{cases} \tag{1.2}$$

where $+$ denotes 32-bit modular addition, $\mathbf{f}_t$ is a round-dependent logical function, $\mathbf{k}_t$ is a 32-bit round-dependent constant, and $\mathbf{W}_t$ is a part of the message schedule, given as follows:

$$\mathbf{W}_t = \begin{cases} \mathbf{M}_t, & 0 \le t \le 15 \\ (\mathbf{W}_{t-3} \oplus \mathbf{W}_{t-8} \oplus \mathbf{W}_{t-14} \oplus \mathbf{W}_{t-16}) \lll 1, & 16 \le t \le 79 \end{cases} \tag{1.3}$$

where $\oplus$ denotes the bitwise XOR operation and $\lll$ denotes 32-bit rotation.

The round-dependent logical functions are given as follows:

$$\mathbf{f}_t(\mathbf{x},\mathbf{y},\mathbf{z}) = \begin{cases} \mathbf{Ch}(\mathbf{x},\mathbf{y},\mathbf{z}), & 0 \le t \le 19 \\ \mathbf{Parity}(\mathbf{x},\mathbf{y},\mathbf{z}), & 20 \le t \le 39 \\ \mathbf{Maj}(\mathbf{x},\mathbf{y},\mathbf{z}), & 40 \le t \le 59 \\ \mathbf{Parity}(\mathbf{x},\mathbf{y},\mathbf{z}), & 60 \le t \le 79 \end{cases} \tag{1.4}$$

where $\mathbf{Ch}(\mathbf{x},\mathbf{y},\mathbf{z})$ is the choice function

$$\mathbf{Ch}(\mathbf{x},\mathbf{y},\mathbf{z}) = (\mathbf{x} \wedge \mathbf{y}) \oplus (\neg\mathbf{x} \wedge \mathbf{z}), \tag{1.5}$$

$\mathbf{Maj}(\mathbf{x},\mathbf{y},\mathbf{z})$ is the majority function

$$\mathbf{Maj}(\mathbf{x},\mathbf{y},\mathbf{z}) = (\mathbf{x} \wedge \mathbf{y}) \oplus (\mathbf{x} \wedge \mathbf{z}) \oplus (\mathbf{y} \wedge \mathbf{z}), \tag{1.6}$$

and $\mathbf{Parity}(\mathbf{x},\mathbf{y},\mathbf{z})$ is the parity function

$$\mathbf{Parity}(\mathbf{x},\mathbf{y},\mathbf{z}) = \mathbf{x} \oplus \mathbf{y} \oplus \mathbf{z}, \tag{1.7}$$

where $\neg$ denotes the bitwise NOT operation, $\wedge$ denotes the bitwise AND operation, and $\vee$ denotes the bitwise OR operation.

1.3 Propositional logic

A propositional variable is an atomic proposition which is either true or false. We denote propositional variables by lowercase letters: p, q, x_1, x_2, etc. We denote the set of all propositional variables by $\mathcal{V}$. Propositional formulae include $\mathcal{V}$ and are built inductively using the unary operator $\neg$ and the binary operators $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, and $\oplus$. We denote propositional formulae by lowercase Greek letters: ϕ, ψ, etc.

The set of truth values is $\{0,1\}$. A (partial) valuation v is a (partial) function from the set of propositional formulae to $\{0,1\}$ such that

$$\begin{aligned} v(\neg\phi) &= \begin{cases} 1 & \text{if } v(\phi) = 0\text{, and} \\ 0 & \text{if } v(\phi) = 1 \end{cases} \\ v(\phi \wedge \psi) &= \begin{cases} 1 & \text{if } v(\phi) = 1 \text{ and } v(\psi) = 1\text{, and} \\ 0 & \text{if } v(\phi) = 0 \text{ or } v(\psi) = 0 \end{cases} \\ v(\phi \vee \psi) &= \begin{cases} 1 & \text{if } v(\phi) = 1 \text{ or } v(\psi) = 1\text{, and} \\ 0 & \text{if } v(\phi) = 0 \text{ and } v(\psi) = 0 \end{cases} \\ v(\phi \rightarrow \psi) &= v(\neg\phi \vee \psi) \\ v(\phi \leftrightarrow \psi) &= v((\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)) \\ v(\phi \oplus \psi) &= v(\neg(\phi \leftrightarrow \psi)) \end{aligned}$$

When $v(\phi) = 1$, we say that v is a model for ϕ.

1.3.1 Inference rules

Inference rules are a cornerstone of logic and allow us to prove that a conclusion is true if the premises are true. We give an example of a famous inference rule below:

$$\frac{\phi \to \psi \qquad \phi}{\psi}\,\textbf{Modus ponens}$$

The rule states that if ϕ implies ψ and ϕ is true, then ψ is true. Here, $\phi \to \psi$ and ϕ are *premises*, and ψ is the *conclusion*. Premises always appear above the inference line, while the conclusion appears below the inference line. The *name* of the rule is written to the right.

Derivations are tree-like objects where the conclusion of one inference is the premise of another inference.

1.3.2 Conjunctive Normal Form

A literal is a variable or its negation; thus, p and $\neg p$ are both literals. For convenience, we sometimes write the literal $\neg p$ as $\bar{p}$.

Negation Normal Form (NNF) formulae are formulae containing only conjunctions, disjunctions, and negations of propositional variables. Any propositional formula can be rewritten in NNF by systematically applying the laws of boolean algebra, in particular by rewriting implications, equivalences, and antivalences in terms of negations, conjunctions, and disjunctions, and "pushing" negations down to the propositional variables:

$$\begin{aligned}
\phi \to \psi &\rightsquigarrow \neg\phi \lor \psi, \\
\phi \longleftrightarrow \psi &\rightsquigarrow (\phi \to \psi) \land (\psi \to \psi), \\
\phi \oplus \psi &\rightsquigarrow \phi \longleftrightarrow \neg\psi, \\
\neg(\phi \land \psi) &\rightsquigarrow \neg\phi \lor \neg\psi, \\
\neg(\phi \lor \psi) &\rightsquigarrow \neg\phi \land \neg\psi.
\end{aligned}$$

A clause is a disjunction $l_1 \lor l_2 \lor \ldots \lor l_n$ of literals. Conjunctive Normal Form (CNF) formulae are conjunctions $c_1 \land c_2 \land \ldots \land c_m$ of clauses. We usually denote CNF formulae by uppercase Greek letters: Φ, Ψ, etc. We will sometimes view a CNF formula as a set of clauses and a clause as a set of literals.

Any NNF formula (and, by transitivity, any propositional formula) can be rewritten in CNF by systematically applying the laws of boolean algebra, in particular by distributing $\lor$ over $\land$:

$$\phi \lor (\psi_1 \land \psi_2) \rightsquigarrow (\phi \lor \psi_1) \land (\phi \lor \psi_2) \tag{1.8}$$

In the worst case, rewriting a propositional formula in CNF using the above rules may yield an exponentially large formula in CNF; every time we apply rule 1.8, we potentially double the size of the original formula since ϕ occurs twice in the new formula.

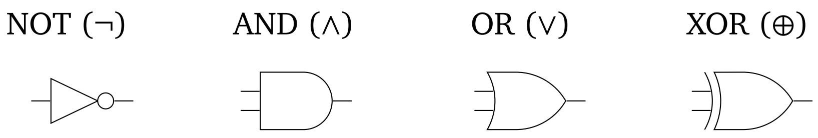


Figure 1.2: Gate types, their names, and corresponding logical connectives.

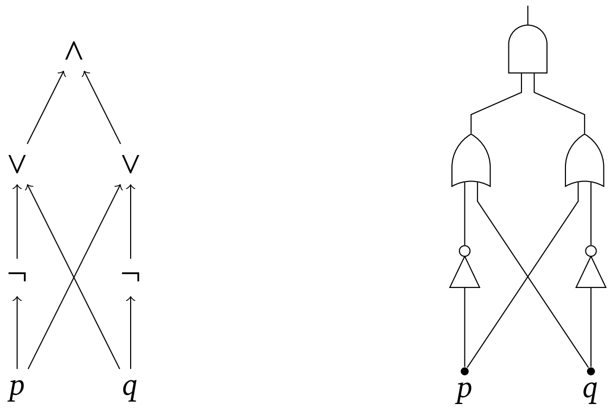


Figure 1.3: Two equivalent drawings of the boolean circuit representing the propositional formula $(\neg p \vee q) \wedge (p \vee \neg q)$; the left drawing shows a graph with (labelled) nodes and edges, while the right drawing shows a circuit diagram.

1.3.3 Boolean circuits

Note that propositional formulae are really just strings of symbols. If we want to include the same subformula more than once, we can use metavariables like ϕ and ψ as abbreviations. However, if we were to spell out a formula that was defined using several nested abbreviations, the fully expanded formula could potentially be very large indeed. We see that even if we had an algorithm that rewrites an arbitrary formula to CNF in linear time, the input formula itself would be too big to be of any use. This is the motivation for the use of *boolean circuits*, a data structure which explicitly allows subformulae to be defined and used more than once.

A boolean circuit is a directed, acyclic graph (DAG) where vertices correspond to propositional connectives (*gates*) or propositional variables (*inputs*) and edges indicate operands (*wires*). We can draw circuits using graphs or using circuit diagrams (Figure 1.2). See Figure 1.3 for the graphical representations of an example boolean circuit.

For a more formal and general treatment of boolean circuits, see [Vollmer, 1999]. Our adaptations include the use of a fixed basis consisting of the boolean operators $\wedge$, $\vee$, $\longleftrightarrow$, and $\oplus$, and including a set of designated wires that act as inverters rather than including negation as a type of gate. Sometimes we will also limit the arity of gates to 2.

1.3.4 Tseitin transformation

We can encode boolean circuits as CNF formulae using the Tseitin transformation [Tseitin, 1968]. Instead of distributing $\vee$ over $\wedge$, we introduce a new variable for each vertex that corresponds to a propositional connective (except negation) and add clauses that define the value of the new variable in terms of its inputs. For the circuit in Figure 1.3, we would introduce the extra variables t_1, t_2, and t_3 and their definitions:

$$t_1 \longleftrightarrow (\neg p \vee q) \qquad\qquad t_2 \longleftrightarrow (p \vee \neg q) \qquad\qquad t_3 \longleftrightarrow (t_1 \wedge t_2)$$

The new definitions are then encoded in CNF using the method described in subsection 1.3.2 (distribution of $\vee$ over $\wedge$). We show the CNF encodings for two such definitions below:

$$\dfrac{\dfrac{\dfrac{\neg t \vee a \qquad \neg t \vee b}{\neg t \vee (a \wedge b)}}{t \rightarrow (a \wedge b)} \qquad \dfrac{\dfrac{\dfrac{\neg a \vee \neg b \vee t}{(\neg a \vee \neg b) \vee t}}{\neg (a \wedge b) \vee t}}{(a \wedge b) \rightarrow t}}{t \longleftrightarrow (a \wedge b)} \qquad\qquad \dfrac{\dfrac{\dfrac{\neg t \vee a \vee b}{\neg t \vee (a \vee b)}}{t \rightarrow (a \vee b)} \qquad \dfrac{\dfrac{\dfrac{\neg a \vee t \qquad \neg b \vee t}{(\neg a \wedge \neg b) \vee t}}{\neg (a \vee b) \vee t}}{(a \vee b) \rightarrow t}}{t \longleftrightarrow (a \vee b)}$$

We note that there exist variations of the Tseitin transformation where only *some* gates are encoded as new variables. In that case, to obtain the definition of a gate, we have to make a depth-first traversal of the graph starting in the given gate (and ending in propositional variables or other gates that should be encoded as new variables). Which gates to encode as new variables can be decided heuristically; for example, one heuristic could introduce new variables only for gates which are used more than once (gates which have an in-degree greater than 1).

1.3.5 Resolution proofs

The resolution rule states that given two clauses containing literals of opposite polarity, we can obtain a new clause (the *resolvent*) containing all the literals from the two clauses except the two literals of opposite polarity. A schema for the inference rule is given as follows:

$$\dfrac{x \vee a_1 \vee a_2 \vee \ldots \vee a_n \qquad \neg x \vee b_1 \vee b_2 \vee \ldots \vee b_m}{a_1 \vee a_2 \vee \ldots \vee a_n \vee b_1 \vee b_2 \vee \ldots \vee b_m}\ \textbf{Resolution}$$

Resolution is a proof system which uses (only) the resolution rule. A resolution proof π of ϕ given Φ, where ϕ is a clause and Φ is a set of clauses, is a sequence of clauses such that each clause is either a clause in Φ or a clause obtained by applying the resolution rule to two clauses which appear earlier in the sequence, and the last clause in the sequence is ϕ. We call π a proof of unsatisfiability of or a *refutation* of Φ if the last clause in the sequence is the empty clause.

If we add the restriction that each resolvent (derived clause) can only be used once, we call the proof system *tree resolution*. Tree resolution is strictly weaker than unrestricted resolution [Rossi et al., 2006], in the sense that the shortest tree resolution proof of a formula is longer than the shortest unrestricted resolution proof of the same formula, since the same clause may have to be derived more than once.

See Figure 1.4 for a visual example of a (tree) resolution proof.

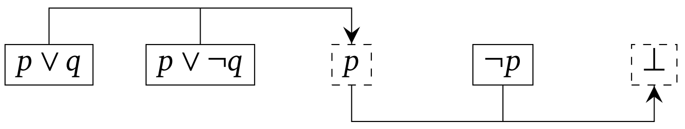


Figure 1.4: An example of a (tree) resolution proof that the CNF formula $(p \vee q) \wedge (p \vee \neg q) \wedge \neg p$ is unsatisfiable.

1.3.6 Pseudo-boolean constraints

A pseudo-boolean constraint (sometimes called linear zero-one constraint or simply linear inequality) in normal form is a constraint of the form

$$c_1 l_1 + \ldots + c_n l_n \geq k,$$

where $c_1, \ldots, c_n$ (the *weights*) and k (the *threshold*) are positive integers, and $l_1, \ldots, l_n$ are literals. We expand the notion of a valuation to a valuation for pseudo-boolean constraints v by requiring that the following also holds:

$$v(c_1 l_1 + \ldots + c_n l_n \geq k) = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} c_i \cdot v(l_i) \geq k \text{ holds, and} \\ 0 & \text{otherwise.} \end{cases}$$

For convenience, we will sometimes use negative weights when writing pseudo-boolean constraints. We can show, using simple algebra, that these constraints can always be rewritten in normal form:

$$\begin{array}{c} cx + \ldots \geq k + c \\ \hline -c + cx + \ldots \geq k \\ \hline -c(1 - x) + \ldots \geq k \\ \hline -c\bar{x} + \ldots \geq k \end{array}$$

Inverted constraints (using $\leq$ instead of $\geq$) can similarly be rewritten by multiplying both sides by -1 and applying the rule for negative weights:

$$\begin{array}{c} -c_1 l_1 - \ldots - c_n l_n \geq -k \\ \hline c_1 l_1 + \ldots + c_n l_n \leq k \end{array}$$

Furthermore, equalities can always be written as a conjunction of two pseudo-boolean constraints in normal form (after applying the rules for inverted constraints and negative weights):

$$\begin{array}{c} c_1 l_1 + \ldots + c_n l_n \leq k \qquad c_1 l_1 + \ldots + c_n l_n \geq k \\ \hline c_1 l_1 + \ldots + c_n l_n = k \end{array}$$

```
function DPLL($\Phi$)
    for all literals $l$ where $l \in \Phi$ do                ▷ Unit propagation
        Remove clauses containing $l$                          ▷ Remove satisfied clauses
        Remove all occurrences of $\neg l$                     ▷ Remove falsified literals
    end for
    if $\Phi = \emptyset$ then                                  ▷ Satisfying valuation
        return SAT
    end if
    if $\emptyset \in \Phi$ then                                ▷ Empty clause
        return UNSAT
    end if
    $x \leftarrow p$ where $p$ is a variable in $\Phi$          ▷ Pick branching literal
    if DPLL($\Phi \cup \{x\}$) = SAT then                       ▷ Branch
        return SAT
    end if
    if DPLL($\Phi \cup \{\neg x\}$) = SAT then                  ▷ Branch
        return SAT
    end if
    return UNSAT
end function
```

Figure 1.5: The original DPLL algorithm. Φ is the CNF formula we are trying to satisfy.

1.4 SAT solvers

Defined loosely, SAT solvers are programs that search for a model for a given formula. Formulae are usually given in CNF, although some solvers additionally also support pseudo-boolean constraints (using the OPB format) or XOR constraints. We can classify a solver as either complete or incomplete. Complete solvers are guaranteed to (eventually) terminate, regardless of whether a model exists or not (and some solvers will even output a resolution proof of unsatisfiability). Incomplete solvers, on the other hand, will only terminate when a model is found; if no model exists, then the solver will never terminate. The difference between the two classes of solvers is that complete solvers are based on a systematic (exhaustive) search, while incomplete solvers are based on stochastic local search (SLS) and have no way to know when all possibilities have been tried.

There are several different algorithms which yield complete solvers. For example, one could simply try all combinations of assignments to the variables in a particular order, essentially constructing the formula's truth table one row at a time. If no row is true, then the formula is unsatisfiable. Of course, the truth table has 2^n rows, where n is the number of variables, so checking all rows may not be feasible in practice.

Variants of Gilmore's algorithm [Gilmore, 1960] transform a CNF formula to a DNF formula by "multiplying" clauses (i.e. distributing $\wedge$ over $\vee$) and shortening the resulting conjunctions by removing duplicate literals and removing conjunctions which contain opposite literals. When there are no more clauses, any conjunction in the DNF formula represents a model of the original formula.

Gentzen's sequent calculus for propositional logic [Buss, 1998] is a form of natural deduction since connectives are analysed in a top-down manner; i.e. the outermost connective is analysed first. It is a search procedure that branches on disjunctions and stops when there are no more connectives to be analysed. Leaves in the search tree represent models of the original formula if they do not contain any opposite literals (contradictions). As pointed out in [D'Agostino, 1992], algorithms based on sequent calculus which do not implement the cut rule are exponential in the number of connectives of the input formula.

1.4.1 The DPLL algorithm and modern CDCL solvers

The foundation for most complete solvers is the DPLL algorithm [Davis and Putnam, 1960; Davis et al., 1962]. The DPLL algorithm branches on each variable in turn by assigning a value to it (a *decision*) and applying unit propagation until fixpoint (see Figure 1.5). Unit propagation is the process of discovering unit clauses (clauses where all literals are falsified except one which is undefined) and augmenting the current valuation in the only way that will satisfy all unit clauses. A conflict occurs when all literals in a clause have been falsified. When a conflict is detected, the algorithm backtracks to the last decision and instead tries the opposite value for the chosen variable or, if both possibilities have already been tried, backtracks even further.

Modern SAT solvers differ from the relatively simple DPLL algorithm in many ways. Firstly, they employ highly specialised algorithms and data structures in order to make unit propagation and backtracking as efficient as possible. One of these specialised algorithms is the *two-watched literal scheme* [Moskewicz et al., 2001], which allows one to detect conflicts as soon as possible without having to visit every clause where a literal occurs. More specifically, the two-watched literal scheme works by maintaining, for each literal, a list of clauses which must be visited when the literal in question is falsified. Not every clause that contains the given literal is on that literal's list; as long as a clause still contains two undefined literals, we know that it cannot propagate or cause any conflicts. The trick is therefore to select two (undefined) literals from each clause which are the *watched* literals. Only when one of these two literals are falsified do we need to visit the clause during unit propagation to find a new watched literal (or, if none can be found, check if the clause is already satisfied or propagate the last undefined literal). The two-watched literal scheme also does not incur any cost when backtracking, since the watches and watchlists remain valid when variables are undefined.

Secondly, modern SAT solvers analyse conflicts to learn new clauses which represent the *reasons* why a particular partial assignment can never be extended to a model of the formula. This gives rise to the name conflict-driven clause-learning (CDCL) solvers. Conflict analysis in the context of SAT solving was first described in [Marques-Silva and Sakallah, 1999]. Put simply, a conflict happens when the solver was unable to detect a falsifying partial valuation through unit propagation alone. The result of conflict analysis is a clause (called a *learnt* clause) which would have prevented the conflict if the solver had known it; i.e. the conflict clause forces the propagation of some literal so that it has the only value which could possibly satisfy the instance under the current partial valuation. Conflict analysis also allows the solver to backtrack further than just

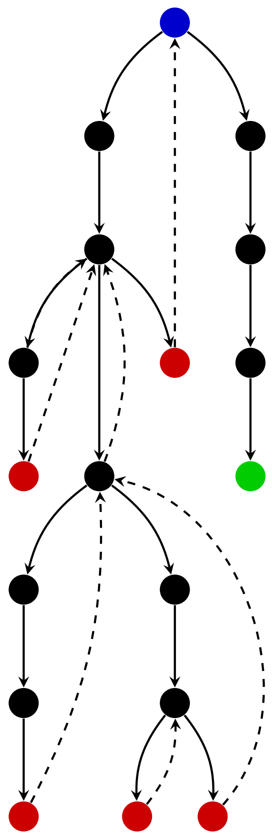


Figure 1.6: An example of search tree for a CDCL solver. The blue node is the root; red nodes denote conflicts; and the green node denotes a solution. Dashed lines indicate backjumping.

the previous decision (see Figure 1.6).

Conflict analysis works by looking at the *implication graph* of a conflict. By definition, we know that the solver tried to propagate a variable in both polarities. If we make the solver keep track of which clauses that caused each variable to be propagated, we see that we end up with a DAG where nodes represent clauses (implications and decisions) and edges represent propagations. Each clause in the graph contains exactly one satisfied or undefined literal (the implied literal for implications and the decision literal for decisions); the rest are falsified. Consequently, any pair of clauses which are connected by an edge contain opposite literals and can be resolved with each other. By recursively resolving all clauses starting from the clause which propagated the conflict in a breadth-first manner until we get to the first *unique implication point* (1UIP; the first point through which all paths from the conflict pass), we obtain a new clause which will keep the solver from entering the same search space (the same partial valuation) again.

The newly learnt clause also gives us the point to which we must backtrack (or backjump, since it may skip more than one decision). It was proved in [Audemard et al., 2008] that the 1UIP scheme is optimal in terms of how far the solver can backtrack. At this point, the clause is unit and the solver is forced to carry out an implication that it previously (without the learnt clause) would not have carried out.

Thirdly, modern CDCL solvers employ fine-tuned heuristics for branching, such as e.g. the VSIDS (Variable-State Independent Decaying Sum) heuristic, simplification passes, and search restarts, etc.

1.5 Algebraic cryptanalysis

Algebraic cryptanalysis is a collection of methods that uses the manipulation of symbolic representations of cryptographic primitives according to laws of algebra. Usually, this means specifying the cryptographic primitive in question as a system of equations (encoding) and finding solutions to the system (solving) [Bard, 2009]. This is in contrast to other well-established methods of cryptanalysis, such as differential and linear cryptanalysis, which are probabilistic in nature.

Many algebraic attacks encode the cryptographic primitive in question as a set of polynomial equations over **GF**(2). In other words, each side of the polynomial equation is a sum (modulo 2) over products of variables and the constants 0 and 1. We give a very simple example of such a system:

$$s = x + y + c_{in}$$
$$c_{out} = xy + xc_{in} + yc_{in}$$

Here, x, y, c_{in}, s, and c_{out} are variables, $+$ denotes addition modulo 2 (i.e. XOR), $\cdot$ denotes multiplication (i.e. conjunction), and the *degree* of the system is 2 (since no product has more than 2 factors). Of course, systems representing real-world ciphers will have thousands of variables and equations. When the system is such that the right-hand side of each equation is 1, the system is said to be in algebraic normal form (ANF).

Although solving these systems of equations is an NP-hard problem [Bard, 2007], some systems which exhibit a certain hidden algebraic structure (e.g. polynomials of low degree) can be solved efficiently in practice using Gröbner bases (e.g. the F4 and F5 algorithms [Faugère, 1999, 2002]) or a combination of linearisation and Gaussian elimination (e.g. the XL algorithm [Bard, 2009] and its variants [Ding et al., 2008]). These techniques are used in AIDA [Vielhaber, 2007] and Cube attacks [Dinur and Shamir, 2008].

1.5.1 Logical cryptanalysis

The term "logical cryptanalysis" was coined in [Massacci and Marraro, 1999] and refers to the use of propositional logic to encode and solve cryptographic primitives; in particular, the system of equations specifying the cryptographic primitive is expressed in CNF, and a SAT solver is used to find solutions of the said system.

In Table 1.1 we list reports of the use of SAT solvers to break the security of cryptographic primitives found in the literature. In most cases, the SAT-based attack is not successful. Notable exceptions to this are [Mironov and Zhang, 2006] and [Courtois et al., 2008]. In [Mironov and Zhang, 2006], the authors encode collision attacks on MD4, MD5, and SHA-0. However, in addition to encoding the cryptographic algorithm itself, they further constrain the resulting system of equations using *differential paths* (from differential cryptanalysis). Differential paths are conditions on the message and internal state bits which hold with a certain non-negligible probability. Differential paths apply only to collision attacks, however.

In [Courtois et al., 2008], the authors encode a known-plaintext, key recovery attack against the block cipher KeeLoq. The crucial idea behind this attack is to exploit a weakness in the algorithm that causes the round-dependent keys to repeat predictably within the key schedule. This knowledge is taken advantage of by encoding the repeating keys using extra constraints.

We find it interesting that both of these successful SAT-based attacks were made possible using extra constraints in addition the original instance. Additionally, the extra constraints are not known to hold with probability 1, so the attack does not necessarily find all possible solutions (or any solution at all). Apart from these two attacks, all other successful SAT-based attacks seem to be made against stream ciphers.

With regards to the SAT solvers used to carry out the attacks, there is a clear preference towards using **MiniSat** and (to a lesser degree) **zChaff**. In [Massacci, 1999], the authors compare the SLS solver **Walk-SAT** with the DPLL-based **Rel-SAT** and find that the SLS solver is definitely worse. We find no subsequent accounts of attempts to use SLS solvers. In [Mironov and Zhang, 2006], the authors test **MiniSat**, **BerkMin**, **Siege**, and **zChaff**, and find **MiniSat** (with preprocessing) to be the best solver. In **Semenov2011a, Semenov2011b, Semenov2011c**, we find the first accounts of experiments using parallel solvers on a cluster.

Table 1.1: Overview of published literature on SAT-based cryptanalysis. We cannot guarantee that the list is complete, however we are fairly sure that it covers the most widely cited papers. Only papers were a SAT solver was used are listed.

Paper		Primitive	Attack
1999	Massacci and Marraro	DES	key recovery
1999	Massacci	DES	key recovery
2000	Massacci and Marraro	DES	key recovery
2003	Fiorini et al.	RSA	signature forgery
2005	Jovanović and Janičić	MD4; MD5	preimage
2006	Mironov and Zhang	MD4; MD5; SHA-0	collision (differential)
2007	De et al.	MD4; MD5	preimage
2007	Srebrny et al.	RSA	factorisation
		SHA-1	preimage
2007	McDonald et al.	Bivium	key recovery
2007	Courtois and Bard	DES	key recovery
2008	Rivest et al.	MD6	preimage
		MD6	collision
2008	Eibach et al.	Bivium	key recovery
2008	Courtois et al.	KeeLoq	key recovery (slide attack)
2008	Chen	Bivium	key recovery
2009	Courtois et al.	Hitag2	key recovery
2009	Erickson et al.	SMS4	key recovery
		S-SMS4	key recovery
2009	Soos et al.	Bivium; Crypto-1; Hitag2; Trivium	state recovery
2009	Renauld and Standaert	PRESENT	(side channel) key recovery
2009	Renauld et al.	AES	(side channel) key recovery
2009	McDonald et al.	SHA-1	collision (differential)
2010	McDonald	Bivium; Trivium	state recovery
2010	Soos	Bivium; Grain; Hitag2; Trivium	state recovery
2010	Gwynne	AES	–
2010	Béjar et al.	RSA	factorisation
		DSA	discrete logarithm
2011a	Semenov et al.	A5/1	(multiple) key recovery
2011	Ignatiev and Semenov	A5/1	key recovery
2011b	Semenov et al.	A5/1	(multiple) key recovery

Chapter 2

Encodings

The first step in any attack using SAT solvers is to obtain a propositional formula that *encodes* the attack.

In general, an encoding is the way in which some problem in a higher-level language is modelled as a propositional problem, i.e. it specifies how to translate higher-level variables and constraints into propositional variables and constraints. In our case, an encoding is a particular way of translating the SHA-1 preimage attack to a CNF formula (sometimes also including XOR and pseudo-boolean constraints).

There is often more than one way to encode a particular (sub-)problem, each of which has its own performance characteristics. For example, one encoding may have many propositional variables, but few clauses or vice versa; another encoding may have few propositional variables and few, but long clauses. The encoding also influences the running time of the SAT solver, perhaps even in different ways for different SAT solvers. Not all encodings are obvious, and some creativity may be needed to discover encodings which allow the problem to be solved efficiently using SAT solvers. For an introduction to encodings of constraint satisfaction problems (CSPs) to SAT problems, see [Walsh, 2000]. Recently, the hardness of different encodings of boolean functions has been investigated in [Gwynne and Kullmann, 2011].

Once we have an encoding of SHA-1 itself, the encoding of a preimage attack consists of simply fixing the bits of the hash to the value that we want to find a preimage for. Assuming that the encoding is sound, any solution to the resulting instance will contain the message that hashes to the given value.

An important property that some encodings have is *arc consistency*. In the context of constraint satisfaction problem (CSP) solving, arc consistency has a very precise definition, however, in this thesis we will use the informal definition given in [Eén and Sörensson, 2006]: “Simply stated, arc-consistency means that whenever an assignment could be propagated on the original constraint, the SAT-solver’s unit propagation, operating on our *translation* of the constraint, should find that assignment too”. In other words, if we e.g. had an arc-consistent encoding of SHA-1, the SAT solver could never make a decision that lead to a conflict, since any variable which would necessarily have a particular value would always be implied through unit propagation. Arc-consistency is obviously a desirable property, but arc-consistent encodings could be exponential in the number of variables [Eén and Sörensson, 2006].

2.1 Circuit (functional) vs. imperative encodings

The most obvious encoding for the compression function of SHA-1, as many other cryptographic algorithms taking fixed numbers of bits as input and output, is to use the Tseitin transformation on the circuit implementing the algorithm (see subsection 1.3.4). In [Massacci and Marraro, 2000], the authors considered this approach for attacks on the block cipher DES: "The straightforward approach would be describing the VLSI circuit implementing DES and transforming the circuit into a logical formula. Unfortunately, the resulting formula is too big to be of any use."

We see that there are at least two possible approaches to the encoding process, namely (1) using the Tseitin transformation on a circuit representation (the circuit or functional encoding), and (2) giving an explicit encoding of higher-level variables and constraints (the imperative encoding). In the remainder of this section, we outline both methods and their advantages/disadvantages; the following two sections describe how the encodings differ for the various constraints used in SHA-1.

Circuit (functional) encoding Despite the apparent drawbacks of constructing and encoding the circuit directly using the Tseitin transformation, this has been the preferred method of encoding hash functions in the literature (e.g. MD4 and MD5 [Jovanović and Janičić, 2005; Mironov and Zhang, 2006; De et al., 2007], MD6 [Rivest et al., 2008], and SHA-1 [Morawiecki and Srebrny, 2010]).

In [Jovanović and Janičić, 2005], the authors advocate the use of their so-called "uniform encoding". Using this method, a circuit is automatically constructed from a high-level description of the hash algorithm using C++ operator overloading. They define a new class (called `Word` in their paper) which, instead of holding a concrete value represented by a sequence of bits, holds an unrealised boolean value represented as a boolean circuit. The operators for `Word` then, rather than compute concrete values, construct circuits from their arguments (see the example in Figure 2.1).

We call this encoding *functional*, since the algorithm in question is represented as a circuit; the input to (resp. output of) the algorithm coincides with the input to (resp. output of) the circuit. The construction of the circuits for various operators is relatively straightforward; bitwise operators map trivially to the corresponding gate types. The only operator which is used in SHA-1 and requires extra explanation is modular 32-bit addition.

One variant of the circuit encoding restricts the set of gate types used in the construction of the circuit (i.e. the *basis* in the terminology of [Vollmer, 1999]). For example, we know from boolean algebra that every boolean function can be expressed in terms of $\neg$ and $\wedge$ only, since $p \vee q \equiv \neg(\neg p \wedge \neg q)$, $p \leftrightarrow q \equiv \neg(\neg p \wedge q) \wedge \neg(p \wedge \neg q)$, etc. In fact, restricting the gates to $\neg$ and $\wedge$ gives us what is called an *and-inverter graph* (AIG).

And-inverter graphs can be useful because they are a conceptually simple data structure that lends itself to optimisation algorithms that attempt to minimise the number of nodes without changing the function computed by the circuit [Brummayer and Biere, 2006]. We may be able to use such algorithms on an AIG representation of SHA-1.

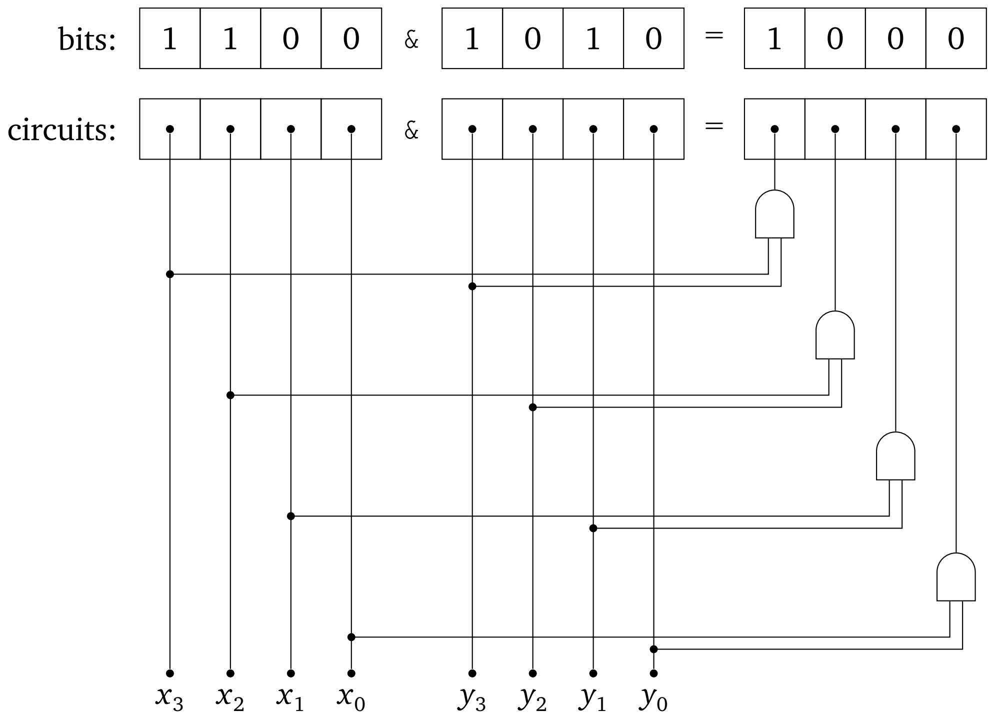


Figure 2.1: The difference between operators on sequences of bits and operators on circuits. Here showing the 4-bit bitwise AND operator.

Imperative encoding An imperative encoding is an encoding which explicitly states how to translate each high-level variable and each constraint into propositional logic. Although more laborious to specify and highly specific to the primitive in question, it also allows finer control over the encoding process.

In fact, the Tseitin transformation is frequently mentioned as a weak point of the the studies found in the literature: In [Jovanović and Janičić, 2005], the authors state: "We are planning to further investigate alternative ways for transforming obtained formulae to CNF (apart from Tseitin's approach) and investigate a possible impact of this on the hardness of generated formulae." Similarly, in [Mironov and Zhang, 2006], the authors state: "For clausification, currently we use the straightforward Tseitin transformation on the propositional formulas. We believe that more optimization can be obtained from careful examination of the encoding process."

Since we find no accounts in the literature of hash functions being encoded in CNF without constructing a big circuit and encoding it in CNF using the Tseitin transformation, we consider this an interesting topic for further research.

2.2 High-level constraints

In this section we will describe some high-level (i.e. non-clausal) constraints which we will use in the encoding of the SHA-1 compression function. Some of these constraints may be accepted directly by the solver or, if not, further encoded in CNF. Where possible, we will discuss several encodings for each constraint.

2.2.1 XOR constraints

An XOR constraint is a formula that consists only of $\oplus$ (XOR) and literals, i.e.: $l_1 \oplus \cdots \oplus l_n$. We call n the length of the constraint. There are two equations in the specification of SHA-1 where we have XOR constraints: the message schedule (Equation 1.3) and one of the round-dependent logical functions (Equation 1.7).

The most straightforward way to encode an XOR constraint with n literals is to transform it to CNF using the methods outlined in subsection 1.3.2. We see that we always end up with a conjunction of 2^{n-1} clauses, where each clause contains all the n variables of the XOR constraint (but as literals with different polarities). For example, to encode the XOR constraint $p \oplus q \oplus r$, we need the following $2^{3-1} = 4$ clauses (each of which is a logical consequence of the original XOR constraint):

$$p \vee q \vee r \qquad p \vee \neg q \vee \neg r \qquad \neg p \vee q \vee \neg r \qquad \neg p \vee \neg q \vee r$$

Since the number of clauses is exponential in the length of the XOR constraint, it may be advantageous to find an alternative encoding for very long XOR constraints. If we introduce extra variables, we can "cut" the XOR constraint into several smaller constraints [Bard et al., 2007]. For example, an XOR constraint with n literals can be cut into two smaller XOR constraints with $1 + \lceil n/2 \rceil$ and $1 + \lfloor n/2 \rfloor$ literals, respectively. Thus, by introducing one extra variable, we can encode an XOR constraint of 20 literals using 2^{11} clauses intead of 2^{19} clauses. At the expense of introducing even more extra variables, we can cut the constraint multiple times and use even fewer clauses. This approach is explored further in [Bard et al., 2007], where they find that the optimal "cutting number" (the number of literals per new XOR constraint) is 6. Consequently, XOR clauses with 6 or fewer literals should be encoded directly without introducing any extra variables.

Some solvers can handle XOR constraints in addition to regular disjunctive clauses. In [Soos et al., 2009], the authors extend the watched literal scheme (for efficient unit propagation) to XOR constraints. The advantage is obvious: the solver only needs to store and perform unit propagation for the equivalent of 1 regular clause, rather than the exponential number of clauses (or extra variables) needed to encode a single XOR constraint. For instances with many/long XOR constraints, this can make a big difference in the running time of the solver.

Additionally, in [Soos et al., 2009; Soos, 2010], the authors extend the DPLL/CDCL algorithm to learn new facts by using Gaussian elimination on the linear system of equations that the XOR constraints form. While gaussian elimination is a polynomial algorithm, their approach requires that it is executed at each decision in the search tree (at least until a certain depth), which quickly becomes quite expensive. In their first implementation, they found that Gaussian elimination gave only a 5% speedup, however, a later improvement to the algorithm gave up to 29% faster solving times.

The approach used in [Soos et al., 2009] for conflict analysis was to treat XOR clauses as the regular disjunctive clause that propagated a literal. For example, for the XOR constraint $p \oplus q \oplus r$ and a partial valuation v where $v(p) = 0$ and $v(q) = 1$, the reason for propagating $v(r) = 0$ is the clause $p \vee \neg q \vee \neg r$. The problem with this approach is that traditional conflict analysis can only learn one disjunctive clause at a time. Recently, there has also been research into extending conflict analysis to *learn* XOR

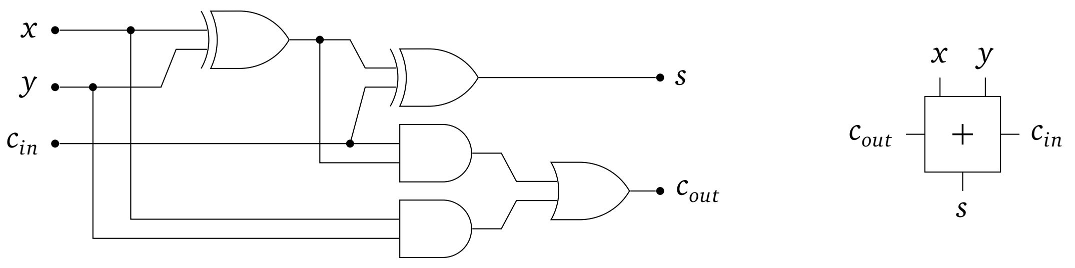


Figure 2.2: A circuit diagram for the full-adder circuit. Here, x, y, and c_{in} (the *carry in*) are the three bits to be summed, while s is the lower bit of the sum and c_{out} (the *carry out*) is the upper bit of the sum. On the left we show the full circuit, and on the right, we give an abbreviation of it.

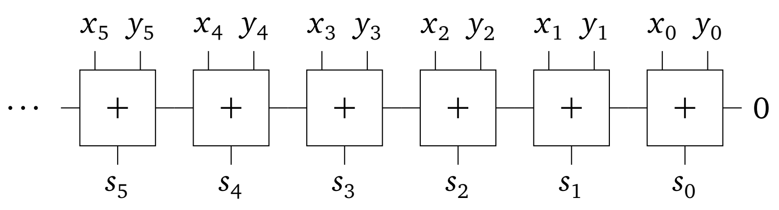


Figure 2.3: A circuit diagram for the ripple-carry adder circuit. The ripple-carry adder is built using a chain of full-adder circuits; c_{in} of bit 0 is wired to 0, while c_{in} of bit i is wired to c_{out} of bit $i+1$. The very last c_{out} is discarded in modular-arithmetic adders.

constraints [Laitinen et al., 2012]. By extending conflict analysis to XOR constraints so that conflict analysis can learn XOR constraints where possible, we see that learning becomes exponentially more powerful in the best case; e.g. instead of learning 2^{n-1} clauses in 2^{n-1} conflicts, we can learn a single XOR constraint with n literals from a single conflict.

2.2.2 32-bit modular addition

Addition is frequently used in cryptographic primitives because it provides diffusion and non-linearity, while being cost-effective; this being a fundamental and frequently executed operation, most CPUs have very fast adders. While the adder circuit is not very difficult to understand conceptually, encoding it in such a way that a SAT solver can handle it efficiently is not straightforward.

A full-adder circuit takes three inputs, x, y, and c_{in} (the three bits to be summed), and outputs two bits, s (the lower bit of the sum) and c_{out} (the *carry* bit). We can give s and c_{out} as functions as follows:

$$\begin{aligned} s(x,y,c_{in}) &= x \oplus y \oplus c_{in} \\ c_{out}(x,y,c_{in}) &= ((x \oplus y) \wedge c_{in}) \vee (x \wedge y) \end{aligned}$$

The Tseitin transformation (or a variant thereof), described in subsection 1.3.4, can be used to obtain a CNF representation of the circuit. Apart from the variables encoding

x, y, and s, we have to introduce 4 extra variables per full-adder, corresponding to the intermediate gates (including c_{out}, but excluding s). Each of the binary XOR gates requires 4 clauses to encode, while each of the binary AND and OR gates requires 3 clauses. Encoding a binary 32-bit adder circuit therefore requires 32×4 variables (in addition to the 3×32 variables for x, y, and s) and $32 \times (2 \times 4 + 3 \times 3) = 544$ clauses.

From [Eén and Sörensson, 2006], we have a different clausification of the full-adder circuit that introduces an extra variable only for the carry bit:

$$\begin{array}{rrrr}
x \vee y \vee c_{in} \vee s & \neg x \vee \neg y \vee \neg c_{in} \vee \neg s & \neg y \vee \neg c_{in} \vee c_{out} & y \vee c_{in} \vee \neg c_{out} \\
x \vee \neg y \vee \neg c_{in} \vee s & \neg x \vee y \vee c_{in} \vee \neg s & \neg x \vee \neg c_{in} \vee c_{out} & x \vee c_{in} \vee \neg c_{out} \\
\neg x \vee y \vee \neg c_{in} \vee s & x \vee \neg y \vee c_{in} \vee \neg s & \neg x \vee \neg y \vee c_{out} & x \vee y \vee \neg c_{out} \\
\neg x \vee \neg y \vee c_{in} \vee s & x \vee y \vee \neg c_{in} \vee \neg s & &
\end{array}$$

This clausification follows from a straightforward conversion of the formula $(s \longleftrightarrow x \oplus y \oplus c_{in}) \wedge (c_{out} \longleftrightarrow ((x \oplus y) \wedge c_{in}) \vee (x \wedge y))$ to CNF. We see that we need 32 extra variables and $32 \times 14 = 448$ clauses to encode a 32-bit adder circuit using this method. This is almost certainly better than the circuit encoding, which requires both more variables and more clauses.

By chaining several full-adder circuits, we obtain what is called a *ripple-carry adder* (see Figure 2.3). This is the simplest type of adder circuit for k-bit integers. However, when implemented as a hardware circuit (and especially when k is large), the physical signal propagation delay for the upper bits of the output may be greater than the desired period of the clock signal (i.e. the circuit is too deep; the longest path between an input and an output is too long). There exist other types of adders which were designed to overcome these problems, generally called *carry-lookahead adders*. An investigation of the performance of different types of adder circuits was made in [Brady and Yang, 2006], however the results were inconclusive (i.e. there was no significant difference between the types of adders).

It is well known that adder networks (adders built using half- and full-adders) do not preserve arc-consistency [Wedler et al., 2004; Eén and Sörensson, 2006]; in other words, under certain partial valuations, there exist possible propagations which are not detected by the SAT solver. These undetected propagations will eventually slow down the SAT solver as it will make decisions which can never lead to a solution and has to invoke conflict analysis and store the conflict clause to prevent the same wrong decision from being made again.

The problem gets worse with more and wider operands: a binary k-bit ($k \geq 2$) ripple-carry adder encoded using the Tseitin transformation has $4 \times k - 8$ "hidden" clauses, i.e. clauses which must be learnt through conflict propagation. For adders with more operands, the problem is even more severe. For a 4-bit ternary ripple-carry adder, there are at least 525 such clauses. These numbers were derived experimentally using a SAT solver modified to output learnt clauses and restarting the search after each conflict.

This problem motivates the search for an encoding of adders that better preserves implicativity. We propose two encodings of n-ary k-bit modular addition based on pseudo-boolean constraints. Pseudo-boolean constraints are theoretically capable of expressing n-ary k-bit modular addition using only a single pseudo-boolean constraint

$$\begin{array}{ccccc} & c_3 & & & \\ & c_2 & c_1 & & \\ & & & c_0 & \\ \hline & x_3 & x_2 & x_1 & x_0 \\ & y_3 & y_2 & y_1 & y_0 \\ + & z_3 & z_2 & z_1 & z_0 \\ \hline = & w_3 & w_2 & w_1 & w_0 \\ \hline\hline \end{array}$$

$$\begin{aligned} x_0 + y_0 + z_0 &= 2c_0 + w_0 \\ c_0 + x_1 + y_1 + z_1 &= 4c_2 + 2c_1 + w_1 \\ c_1 + x_2 + y_2 + z_2 &= 4d_0 + 2c_3 + w_2 \\ c_3 + c_2 + x_3 + y_3 + z_3 &= 4d_2 + 2d_1 + w_3 \end{aligned}$$

Figure 2.5: Grade school (binary) addition schema and corresponding pseudo-boolean constraints for ternary 4-bit addition. $c_0, \ldots, c_3$ are extra variables introduced to encode the *carry bits*. $d_0, \ldots, d_2$ are dummy variables encoding the carry bits which carry beyond the 4-bit result.

(two linear inequalities). Let $\mathscr{F}_{+_{n,k}}(f, x_1, \ldots, x_n)$ be the encoding of $\mathbf{f} = \mathbf{x}_1 + \cdots + \mathbf{x}_n$, where f and $x_1, \ldots, x_n$ encode $\mathbf{f}$ and $\mathbf{x}_1, \ldots, \mathbf{x}_n$, respectively, and $\mathbf{f}$ and $\mathbf{x}_1, \ldots, \mathbf{x}_n$ are sequences of k bits:

$$\begin{aligned} \mathscr{F}_{+_{n,k}}(f, x_1, \ldots, x_n) &= \left(\sum_{i=0}^{k-1} 2^i \cdot f_i = \sum_{i=0}^{k-1} 2^i \cdot (x_{i,1} + \ldots + x_{i,n}) \right) \\ &= \left(\sum_{i=0}^{k-1} 2^i \cdot (f_i - x_{i,1} - \ldots - x_{i,n}) = 0 \right). \end{aligned}$$

Applying the rewriting rules given in subsection 1.3.6, we can obtain two pseudo-boolean constraints in normal form. However, most solvers impose a limit on the values of the weights. Many solvers do not support weights or a threshold greater than $2^{24} - 1$, and many solvers do not support constraints where the sum of the weights (possibly also including the threshold) is greater than $2^{32} - 1$. These limitations are consequences of the fact that most implementations use 32-bit integers to store and handle the weights and thresholds in pseudo-boolean constraints. Some solvers use arbitrary-precision arithmetic; however, this comes at the expense of efficiency when compared to solvers using fixed-width arithmetic.

In order to bypass these limitations, we can introduce a few extra variables and split the constraints into several smaller (shorter) constraints. Looking at Figure 2.2, we see that the role of the adder circuit is essentially that of summing three 1-bit numbers (two operands and one carry-in) and outputting the result as a 2-bit number. Generalising this to more than two operands, we would obtain a circuit that sums n 1-bit numbers and outputs their sum as a $\lfloor 1 + \log_2 n \rfloor$-bit number. The most significant bits of the sum are given as additional inputs to the next adders. These circuits can be encoded using a single pseudo-boolean constraint over $n + \lfloor 1 + \log_2 n \rfloor$ variables; in Figure 2.5 we show a concrete example of how to use multiple such pseudo-boolean constraints to encode ternary 4-bit addition.

This encoding has to our knowledge never before been described in the literature.

2.2.3 Pseudo-boolean constraints

If we use a SAT solver that doesn't support pseudo-boolean constraints directly, we can also translate pseudo-boolean constraints to CNF in an extra step. In [Eén and Sörensson, 2006], the authors construct a circuit that encodes whether the constraint is satisfied or not and force the circuit's output to true. The construction of the circuit is done using three mechanisms: Binary Decision Diagrams (BDDs), adder networks, and sorter networks.

We instead consider a more direct approach. For any constraint over n propositional variables, its truth table has exactly 2^n entries. When n is small, we can enumerate the truth table and use a logic minimiser such as **Espresso** [Rudell and Sangiovanni-Vincentelli, 1987] to minimise the function. The final number of clauses needed to encode the constraint varies depending on the constraint, but is in any case bounded by the size of its truth table.

2.2.4 Unary/binary constraints

As we noted in subsection 2.2.2, the pseudo-boolean constraints used to encode modular addition are essentially conversions between the unary and binary representations of a number (in particular, we are using these constraints to count the number of 1s in a particular column of the adder; each digit of the binary representation of the sum carry over to the next column, except the lowermost bit which becomes the output bit of that position). We can define a new constraint type, a specialisation of pseudo-boolean constraints, that implements these exact semantics: a unary/binary constraint is a constraint of the form

$$l_{n-1} + \cdots + l_0 = r_{m-1} \circ \cdots \circ r_0$$

where $l_0, \ldots, l_{n-1}$ and $r_0, \ldots, r_{m-1}$ are literals. We expand the notion of a valuation to a valuation for unary/binary constraints v by requiring that the following also holds:

$$v(l_{n-1} + \cdots + l_0 = r_{m-1} \circ \cdots \circ r_0) = \begin{cases} 1 & \text{if } \sum_{i=0}^{n-1} v(l_i) = \sum_{i=0}^{m-1} 2^i \cdot v(r_i) \text{ holds, and} \\ 0 & \text{otherwise.} \end{cases}$$

2.3 Encoding of SHA-1

2.3.1 Message schedule

Let $\mathscr{F}_{\mathrm{W}}(f, x, y, z, w)$ denote the encoding of $\mathbf{f} = (\mathbf{x} \oplus \mathbf{y} \oplus \mathbf{z} \oplus \mathbf{w}) \lll 1$, where f, x, y, z, and w encode $\mathbf{f}$, $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$, and $\mathbf{w}$, respectively. We can specify it as follows:

$$\begin{aligned} \mathscr{F}_{\mathrm{W}}(f, x, y, z, w) &= \bigwedge_{i=0}^{31} f_i \longleftrightarrow x_{i \lll 1} \oplus y_{i \lll 1} \oplus z_{i \lll 1} \oplus w_{i \lll 1} \\ &= \bigwedge_{i=0}^{31} \neg f_i \oplus x_{i \lll 1} \oplus y_{i \lll 1} \oplus z_{i \lll 1} \oplus w_{i \lll 1} \end{aligned}$$

where $i \lll 1$ denotes a permutation on the integers $0, \ldots, 31$ applied to i such that

$$i \lll 1 = \begin{cases} 31, & i = 0 \\ i-1, & i \geq 1 \end{cases}$$

(i.e. a $i \lll 1$ denotes left-rotation of the index i by 1). Using this to encode $\mathbf{W}_t$ for $16 \leq t \leq 79$, we obtain $64 \times 32 = 2048$ XOR constraints, each with 5 literals. Since each XOR constraint only has 5 literals, it seems pointless to try to split the constraints into further, smaller constraints.

We can, however, also consider a second approach to encoding the message schedule. Instead of defining each bit of each word $\mathbf{W}_{16}, \ldots, \mathbf{W}_{79}$ of the message schedule in terms of $\mathbf{W}_{t-3}$, $\mathbf{W}_{t-8}$, $\mathbf{W}_{t-14}$, and $\mathbf{W}_{t-16}$, we can substitute the definitions of these words until we can express each bit in $\mathbf{W}_{16}, \ldots, \mathbf{W}_{79}$ only in terms of the bits of $\mathbf{W}_0, \ldots, \mathbf{W}_{15}$. This would result in $64 \times 32 = 2048$ XOR constraints of varying lengths (between 5 and 75; on average 35), but with the same number of variables.

2.3.2 Round-dependent logical functions

Next we describe how to encode the round-dependent logical functions $\mathbf{Ch}(\mathbf{x}, \mathbf{y}, \mathbf{z})$, $\mathbf{Maj}(\mathbf{x}, \mathbf{y}, \mathbf{z})$, and $\mathbf{Parity}(\mathbf{x}, \mathbf{y}, \mathbf{z})$ given in section 1.2.

Let $\mathscr{F}_{\mathbf{Ch}}(f, x, y, z)$ denote the encoding of $\mathbf{f} = \mathbf{Ch}(\mathbf{x}, \mathbf{y}, \mathbf{z})$, where f, x, y, and z encode $\mathbf{f}$, $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$, respectively. Recall that the choice function $\mathbf{Ch}$ is defined as $\mathbf{Ch}(\mathbf{x}, \mathbf{y}, \mathbf{z}) = (\mathbf{x} \wedge \mathbf{y}) \oplus (\neg \mathbf{x} \wedge \mathbf{z})$. We define $\mathscr{F}_{\mathbf{Ch}}(f, x, y, z)$ as:

$$\begin{aligned}
\mathscr{F}_{\mathbf{Ch}}(f, x, y, z) &= \bigwedge_{i=0}^{31} f_i \longleftrightarrow (x_i \wedge y_i) \oplus (\neg x_i \wedge z_i) \\
&= \bigwedge_{i=0}^{31} (\neg f_i \vee \neg x_i \vee y_i) \wedge (\neg f_i \vee x_i \vee z_i) \wedge (\neg f_i \vee y_i \vee z_i) \\
&\qquad \wedge (f_i \vee \neg x_i \vee \neg y_i) \wedge (f_i \vee x_i \vee \neg z_i) \wedge (f_i \vee \neg y_i \vee \neg z_i).
\end{aligned}$$

We note that the choice function is essentially what is called a MUX circuit. It is well known that applying the naïve Tseitin transformation to these circuits will introduce more variables and clauses than are really needed [Eén and Biere, 2005]. Each $\wedge$ gate gives rise to an extra variable and 3 clauses. The $\oplus$ gate will need 4 clauses. The Tseitin transformation would thus need 2 extra variables and 10 clauses, whereas the direct conversion to CNF would need no extra variables and only the 6 clauses shown above.

Let $\mathscr{F}_{\mathbf{Maj}}(f, x, y, z)$ denote the encoding of $\mathbf{f} = \mathbf{Maj}(\mathbf{x}, \mathbf{y}, \mathbf{z})$, where f, x, y, and z encode $\mathbf{f}$, $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$, respectively. Recall that the majority function $\mathbf{Maj}$ is defined as $\mathbf{Maj}(\mathbf{x}, \mathbf{y}, \mathbf{z}) = (\mathbf{x} \wedge \mathbf{y}) \oplus (\mathbf{x} \wedge \mathbf{z}) \oplus (\mathbf{y} \wedge \mathbf{z})$. We define $\mathscr{F}_{\mathbf{Maj}}(f, x, y, z)$ as:

$$\begin{aligned}
\mathscr{F}_{\mathbf{Maj}}(f, x, y, z) &= \bigwedge_{i=0}^{31} f_i \longleftrightarrow (x_i \wedge y_i) \oplus (x \wedge z) \oplus (y \wedge z) \\
&= \bigwedge_{i=0}^{31} (\neg f_i \vee x_i \vee y_i) \wedge (\neg f_i \vee x_i \vee z_i) \wedge (\neg f_i \vee y_i \vee z_i) \\
&\qquad \wedge (f_i \vee \neg x_i \vee \neg y_i) \wedge (f_i \vee \neg x_i \vee \neg y_i) \wedge (f_i \vee \neg y_i \vee \neg z_i).
\end{aligned}$$

Finally, let $\mathscr{F}_{\textbf{Parity}}(f,x,y,z)$ denote the encoding of $\mathbf{f} = \textbf{Parity}(\mathbf{x},\mathbf{y},\mathbf{z})$, where f, x, y, and z encode $\mathbf{f}$, $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$, respectively. Recall that the parity function **Parity** is defined as $\textbf{Parity}(\mathbf{x},\mathbf{y},\mathbf{z}) = \mathbf{x} \oplus \mathbf{y} \oplus \mathbf{z}$. We define $\mathscr{F}_{\textbf{Parity}}(f,x,y,z)$ as:

$$\begin{aligned}\mathscr{F}_{\textbf{Parity}}(f,x,y,z) &= \bigwedge_{i=0}^{31} f_i \longleftrightarrow x_i \oplus y_i \oplus z_i \\ &= \bigwedge_{i=0}^{31} \neg f_i \oplus x_i \oplus y_i \oplus z_i.\end{aligned}$$

We see that we end up with 32 XOR constraints which may be further encoded in CNF using the method outlined in subsection 2.2.1.

2.3.3 Intermediate state and final hash value

In order to encode Equation 1.1 and Equation 1.2, we need to encode binary and 5-ary addition, respectively. We can do this using any of the methods for encoding addition described earlier.

To encode n-ary k-bit modular addition using pseudo-boolean constraints, we need k pseudo-boolean constraints. Each constraint corresponds to a bit of the output and encodes the relationship between the input bits and output bits at each bit position, including the carry-in and carry-out bits. In order to encode 5-ary 32-bit modular addition, we need 1 constraint of length 8, 1 constraint of length 9, and 30 constraints of length 10. We need to introduce two extra variables for each output bit to represent the carry bits. (Note that we don't need carry bits for the most significant output bits, since they are not used. This fact could be used to reduce the number of variables even further.)

Since no pseudo-boolean constraint (for the binary and 5-ary adders in SHA-1) has more than 10 literals, the whole truth table has at most $2^{10} = 1024$ entries, which is entirely feasible to enumerate. We use the logic minimiser **Espresso** [Rudell and Sangiovanni-Vincentelli, 1987] to minimise the function to 173 clauses (1185 literals). Although we don't have any proof ready, we believe that this encoding of modular addition preserves more implications (i.e. is closer to achieving the goal of arc-consistency) than the Tseitin transformation of an adder circuit.

2.4 Reduced instances

We cannot actually measure the time it would take a SAT solver to break SHA-1, since it would take too long; besides, if we could, we would have no reason to undertake the investigation in the first place. We must therefore always work with *reduced* instances, i.e. instances which are easier to solve, but similar in nature to the full instances that we wish to solve.

There are several ways to reduce the difficulty of the instance. Firstly, we can reduce the number of *rounds* of the SHA-1 algorithm; the full algorithm uses 80 rounds where the message block is the direct input of the first 16 rounds, so possible number of rounds range from 16 to 80. The purpose of each round is to complicate the relationship between the input bits and the output bits, e.g. by removing linearities and making each output bit depend on all the input bits.

Secondly, we can reduce the number of *message bits* by fixing them to particular values. The message bits are the independent variables of the instance and give an upper bound on the size of the search space (a brute force attack would in the worst case have to try all combinations of the message bits[1]). Since SHA-1 operates on fixed message blocks of 512 bits, the possible number of bits to leave as free variables range from 0 to 512.

We additionally have the choice of fixing message bits to random values or to values which we know have at least one solution. If we fix the message bits to random values, we run the risk of creating unsatisfiable instances. While the running time of SAT solvers is known to be more stable on unsatisfiable instances, the SAT solver must be able to output a resolution proof if we want to verify that indeed no solution exists. Another reason to avoid unsatisfiable instances is that they are never encountered in non-reduced instances.

Thirdly, we can reduce the number of *output bits* by *not* fixing them to particular values. That is, we allow some of the output bits to deviate from the particular hash value that we are trying to find a preimage for. Because of the pseudo-random property of the compression function, we expect that a random message block maps to a given hash value with probability 2^{-n}, where n is the number of output bits. We therefore expect to reduce the difficulty of the instance by a factor of 2 for each output bit that remains unfixed. Since the output of SHA-1 is 160 bits, the possible number of output bits to leave unfixed ranges from 0 to 160.

It is not completely unreasonable to expect that certain bits of either the message or the hash will influence the running time more or less than other bits, especially if we also reduce the number of rounds. To mitigate a potential unknown bias due to the choice of which bits are fixed, the choice is made at random for each generated instance.

Soos et al. investigated the effect of fixing ("guessing") the independent bits of a SAT instance encoding a state recovery attack on stream ciphers [Soos et al., 2009] and found that the average running time of the SAT solver changes almost perfectly-exponentially with the number of fixed bits; each additional bit that is fixed reduces the average running time by a factor $2^{-0.7}$.

2.5 Implementation

We have implemented the instance generator in C++. Parameters include the number of rounds and the number of message and hash bits to fix in advance. It can generate instances in both DIMACS CNF and OPB formats. For CNF output, explicit XOR and unary/binary constraints can be enabled optionally (otherwise, XOR constraints of length n are encoded using 2^{n-1} clauses and no extra variables, and unary/binary constraints are encoded using **Espresso** to minimise their truth tables). For OPB

[1]On average, we expect a brute force attack to succeed much sooner, however, since we expect each hash value to have a multitude of inputs that map to it.

output, it is possible to use either compact adders (one pseudo-boolean constraint per adder circuit) or unary/binary constraints (k pseudo-boolean constraints per k-bit adder circuit).

Although the program relies on the external program **Espresso** for minimisation of unary/binary constraints, the whole encoding process is self-contained; i.e. from the perspective of the user, the program only needs to be invoked once from the command line to generate the complete instance.

In addition, two scripts are provided which (1) check that the solution found by a SAT solver indeed satisfies the generated instance (in either CNF or OPB format); and (2) check that the solution found by a SAT solver indeed is a valid preimage. The first script allows us to detect errors in SAT solvers. The second script allows us to detect errors in the encoding process (assuming that any solution found satisfy the generated instance). By using these scripts to check every solution found during experiments, we ensure that our results are valid.

The program and scripts are available as Free Software (under the GNU General Public License version 3) from `https://github.com/vegard/sha1-sat/` or `http://folk.uio.no/vegardno/sha1-sat`.

Chapter 3
Methodology

Our ultimate goal is to investigate how effective SAT solvers can be at breaking the preimage resistance property of SHA-1. Our baseline measure is the efficiency of an off-the-shelf SAT solver on an instance generated using standard techniques. We will try to change the various heuristics of the solver and instance generation process in order to identify the ones that are better. Each such change therefore forms the basis of an experiment.

In the SAT literature and in the SAT solver competitions, the most common way to test the merits of a new solver, technique, or encoding is to solve a relatively large set of benchmark instances with a predetermined per-instance timeout. Both the total number of solved instances and the total time spent on solving are used as indicators of the performance of a solver. However, most SAT papers run the solver on the whole set of instances only once; as such, we have no idea of the *variability* of the total time. Combined with the fact that the running time of the same solver and the same instance can vary quite a lot (up to 2–3 orders of magnitude) [Bard et al., 2007; Eibach et al., 2008] it becomes impossible to tell whether the result presented in the paper is representative of the true performance of the solver or merely a chance result.

It was pointed out in [Nikolić, 2010] and later [Gelder, 2011] that the traditional method of evaluating the performance of SAT solvers is deficient and that the SAT community should move to statistically well-founded methods. In the following sections, we will outline the methodology we have used for the experiments in the rest of the thesis.

3.1 Estimating running time

Because of the big variation in the running time of a SAT solver on the same instance, it is not enough to run the solver just once for each configuration; instead, we have to obtain a *sample* consisting of multiple runs. From a sample we can use statistical procedures to calculate a *confidence interval* that almost surely (e.g. 95% of the time) contains the true value of the parameter that we are estimating.

We also have to choose the *parameter of interest*; we might want to carry out a worst-case analysis by looking at the sample maximum, but this is in general difficult because the sample maximum is not robust with respect to outliers and cannot be used as an estimator of the actual worst-case scenario. For average-case analysis, which is what we will use in this thesis, the parameter of interest is the *expected value* of the running time, for which the sample mean is an unbiased estimator.

As noted in [Bard et al., 2007], and as we shall see in section 4.2, the running time of SAT solvers for at least some cryptographic instances appears to be well approximated by a lognormal distribution. This complicates the procedure for finding confidence intervals for the estimated mean somewhat, since the usual textbook procedure assumes that the sample comes from a normal distribution. Some sources advocate the use of a log-transformation on the lognormal data to obtain a confidence interval which can then be transformed back using exponentiation. However, this will give us a confidence interval for the *geometric* rather than the arithmetic mean.

We consider three approaches for calculating confidence intervals for the estimated mean: (i) We can use the usual procedure for normal data on our untransformed sample. Due to the law of large numbers, the *sample mean* is asymptotically normal in the number of samples, so a sufficient sample size could still give reasonably reliable results; (ii) We can use a bootstrap procedure such as the BC_a procedure advocated in [DiCiccio and Efron, 1996]. In short, bootstrap procedures create new samples using the original sample as an approximation of the underlying distribution; (iii) We can use the approximate method for calculating confidence intervals for the mean assuming a lognormal distribution suggested by Sir David Cox and described in [Land, 1972; Zou et al., 2009]. In section 4.2, we will investigate whether lognormality is a reasonable assumption. Unless specified otherwise, all confidence intervals reported in this thesis will be calculated using the BC_a procedure, since it is expected to perform best when the underlying distribution of the population is unknown or known to be non-normal.

3.2 Comparing configurations

A *configuration* is the combination of a (possibly modified) solver, the parameters of the solver, and the parameters of an encoding. Most of our experiments will be comparisons between two or more configurations. There are two ways to compare such measurements, essentially corresponding to the following questions:

1. Which configurations are the *fastest*?
2. Which configurations *scale* better?

The first question is relatively straightforward to answer; we simply have to compare the two (or more) configurations on the same (reduced) instance. In the methodology of Nikolić [Nikolić, 2010], the parameter of interest is the probability that one configuration finishes before the other when run in parallel. In other words, we say that configuration C_1 is better than configuration C_2 if the probability that C_1 finishes before C_2 is smaller than 0.5 (at a certain significance level). However, this is not equivalent to the mean running time of C_1 being smaller than C_2; in fact, they are independent measures. (To see this, consider the case that C_1 always solves the instance very fast, except that it has a very small probability of taking an extremely long time to finish, while C_2 always solves the instance just a little bit slower than the typical case of C_1. Then C_1 is more likely to solve the instance before C_2, but its mean running time will be greater.)

In order to test that the observed difference between two means is statistically significant, we have to use a statistical test such as e.g. Student's t-test. However, as with the normal procedure for computing confidence intervals, the t-test is not sufficient for our purposes, since it assumes that the underlying distribution is normal. If the t-test is performed on the log-transformed data, this will actually test the median rather than the mean [Zou et al., 2009], and if the variances are unequal, we can end up concluding either that there is not enough evidence to discern any difference when there is in fact a difference (failing to reject a false null hypothesis), or we can end up concluding that there is a difference when in fact there is no difference (falsely rejecting the true null hypothesis) [Zhou et al., 1997]. For samples that we know come from a population that is lognormal-distributed, we can use the procedure described in [Zhou et al., 1997].

If we cannot assume a specific distribution for the population, a straightforward test to see if there is a significant difference between two means is to use the aforementioned BC_a bootstrap procedure to obtain a confidence interval for the difference between the means. If this interval contains zero (no difference), then we do not have enough evidence to reject the hypothesis that the two means are the same. Conversely, if the interval does *not* contain zero, we can reject the hypothesis that the two means are the same.

When working with reduced instances, simply testing the difference between two configuration is not always enough, since it does not necessarily tell us much about the difference for more difficult instances. For example, it could happen that configuration C_1 is faster than configuration C_2 for 20 rounds, but slower for 25 rounds. To answer the second question ("Which configuration *scales* better?"), we can run both configurations on a *range* of difficulties (degrees of reduction) and attempt to extrapolate the pattern to even more difficult instances. However, while we expect e.g. solving time to be approximately exponential in the number of fixed hash bits, it is also perfectly possible that this is not at all the case.

3.3 Multiple comparisons

When performing multiple hypothesis tests (or deriving multiple confidence intervals), the overall confidence level of the analysis will be lower than the confidence level of the individual tests (or intervals). This is because the confidence level expresses the probability of obtaining the significant result by chance; if we construct 100 confidence intervals at the 95% confidence level, we expect approximately 5 of the intervals to fail to contain the true parameter that they estimate. The overall confidence level of the analysis (the confidence that *all* the confidence intervals cover their true means) is therefore much lower than 95%.

For experiments where we make multiple comparisons (e.g. between many different solvers or between many different settings of a solver parameter), we must therefore take into account the overall confidence level as well as the confidence level of the individual tests or estimates. One procedure for testing all pairwise differences between m populations, taking the overall confidence level into account, is Tukey's test. However, Tukey's test makes two assumptions that are unreasonable in our context: firstly, it assumes that the populations are approximately normal; secondly, it assumes that the populations have equal variance. A better procedure for our purposes might be the Games-Howell test [Games et al., 1979], since it does not make the assumption of equal variance.

Another option is to plan which comparisons we want to make before obtaining or inspecting the data (to prevent biasing our decisions of which comparisons to make) and use the so-called Bonferroni correction to adjust the significance level. The Bonferroni correction makes no assumptions about the dependence or independence of the tests, so if some of our tests are dependent, the procedure is not necessarily as powerful as it could have been if we had assumed (some) dependence. If we wish to make m comparisons, the *adjusted significance level* α' is α/m [Abdi, 2007]. Similarly, if we wish to derive m confidence intervals, the *adjusted confidence level* $1-\alpha'$ is $1-\alpha/m$.

3.4 Censoring

Due to the unknown and potentially very long solving times, many studies and SAT solver competitions employ a time limit for the solver. Any sample containing runs where the solver was stopped before a solution was found is said to contain *censored* data. When calculating the mean (and many other statistics), these runs can neither simply be discarded, nor included with the time at which the solver was stopped, since this would tend to underestimate the mean. Instead, the missing samples must be estimated using a combination of assumptions and the other samples. While many statistical procedures have been developed to take censored data into account (as in e.g. the methodology of Nikolić [Nikolić, 2010]), to simplify the analyses in this thesis we will not consider censored data at all, i.e. all experiments will run to completion or not even be considered if the time is too large to obtain a complete sample.

3.5 Data collection

To carry out the experiments, we run the solvers on a cluster of identical machines. Each machine has 8 Intel Core i7-870 CPUs at 2.93GHz with 8 MiB cache and 8 GiB RAM, however we use only one core at a time for any given machine to prevent contention for shared resources like higher-level cache and RAM. Because of the high variability of the running times of SAT solvers, we will usually use a sample size of $n = 100$ runs, using different random seeds for both instance generation and the solver. We will sometimes use smaller sample sizes, in particular when we just wish to get a rough idea of the distribution or when the high running time prohibits collecting more samples. Every solution is verified to satisfy the encoded instance (e.g. the CNF file) and to be an actual preimage by external programs.

All values computed from the running times (such as e.g. means, confidence intervals, etc.) are computed using the statistical package R [R Core Team, 2012].

Chapter 4

Experiments

This chapter contains our experiments and their results. Each section first describes the motivation and purpose of the experiment and how it will be carried out. Then we give the results and a discussion of the results. Our experiments are organised as follows:

Category	Sect.	Description
Preliminaries	4.1	Comparison between 12 SAT solvers
	4.2	Running time distribution of **MiniSat**
	4.3	Running time as a function of the difficulty
	4.4	The effect of fixing specific message/hash bits
Encodings	4.5	Comparison between CNF encodings
	4.6	Comparison between CNF, pseudo-boolean constraints, and unary/binary constraints
	4.7	The effect of XOR constraints and Gaussian elimination
Simplification	4.8	The effect of preprocessing and simplification
Heuristics	4.9	Comparison between branching heuristics; search for the optimal variable activity decay factor
	4.10	Comparison between geometric and Luby restart heuristics; search for the optimal restart interval
	4.11	The effect of conflict clause minimisation and the use of reverse arcs
	4.12	Search for the optimal clause activity decay factor
Learnt clauses	4.13	Running time when searching for multiple solutions
	4.14	The effect of reusing learnt clauses

4.1 SAT solvers

In our first experiment, we will compare the performance of a number of unmodified, freely available SAT solvers and standard encodings. The purpose of this first experiment is twofold; first, to establish a baseline measure with which we can compare subsequent experiments, and second, to pick one or two configurations, our best candidates, which we will modify in the subsequent experiments. We make this selection under the assumptions that (1) any improvement found in the best solver would yield corresponding improvements for other configurations (where applicable), and (2) any improvement found in the best solver would yield corresponding improvements for larger instances. We will use the Games-Howell procedure to test for significant differences between the running times of all the solvers.

The solvers which we would like to test are the following: **glucose** [Audemard and Simon, 2009], **lingeling** [Biere, 2010], **clasp** [Gebser et al., 2007], **PrecoSAT** [Biere, 2010], **Rsat** [Pipatsrisawat and Darwiche, 2007a], **PicoSAT** [Biere, 2010], **MiniSat** [Eén and Sörensson, 2004], **CryptoMiniSat** [Soos et al., 2009], and **Sat4J** [Berre and Parrain, 2010]. These solvers have ranked high in the SAT Competitions and Races of the last few years. We do not test the solvers **SATzilla2012** [Xu et al., 2012] (because we could not find out how to make it output the satisfying solution), **march** [Heule et al., 2005; Heule and van Maaren, 2006] (because it does not solve even comparatively easy instances in reasonable time), or any SLS solver, since we expect them to perform badly on difficult instances like cryptographic problems [Massacci, 1999; Massacci and Marraro, 2000]. Very preliminary tests with **sparrow2011** [Balint et al., 2011] and **ubcsat** [Tompkins and Hoos, 2005] seem to confirm this, as they were unable to solve even a single instance (the best algorithm in **ubcsat** still had around 200 unsatisfied clauses as its best result after running for an hour).

Since we only wish to get a rough impression of the performance of the various solvers and since we cannot predict how long it will take to obtain all the samples, we choose a relatively easy instance with 21 rounds, 0 fixed message bits, and 160 fixed hash bits.

Results

The boxplot in Figure 4.1 shows that the running time varies a lot, both between the solvers (up to 3 orders of magnitude) and for a single solver (up to 4 orders of magnitude).

It appears that **clasp-2.0.6-opb** is the best configuration with a mean running time at 40.48 s, followed by **MiniSat-2.2.0** at 81.14 s, and **clasp-2.0.6-cnf** at 103.84 s, after which follows a plateau of four solvers with means that are quite close to one another (174.32–238.91 s). The worst solver appears to be **glucose-2.0** at 1,148.77 s. We can also see several cases where the mean of one sample is greater than the mean of another while the median is smaller, e.g. between **Minisat-2.2.0** and **clasp-2.0.6-cnf**. This illustrates the importance of not simply testing for significant differences using the t-test on the log-transformed samples.

Table 4.1 lists all the pairwise differences between the solvers. The first thing to notice is that the differences between the best solver (**clasp-2.0.6-opb**) and all the

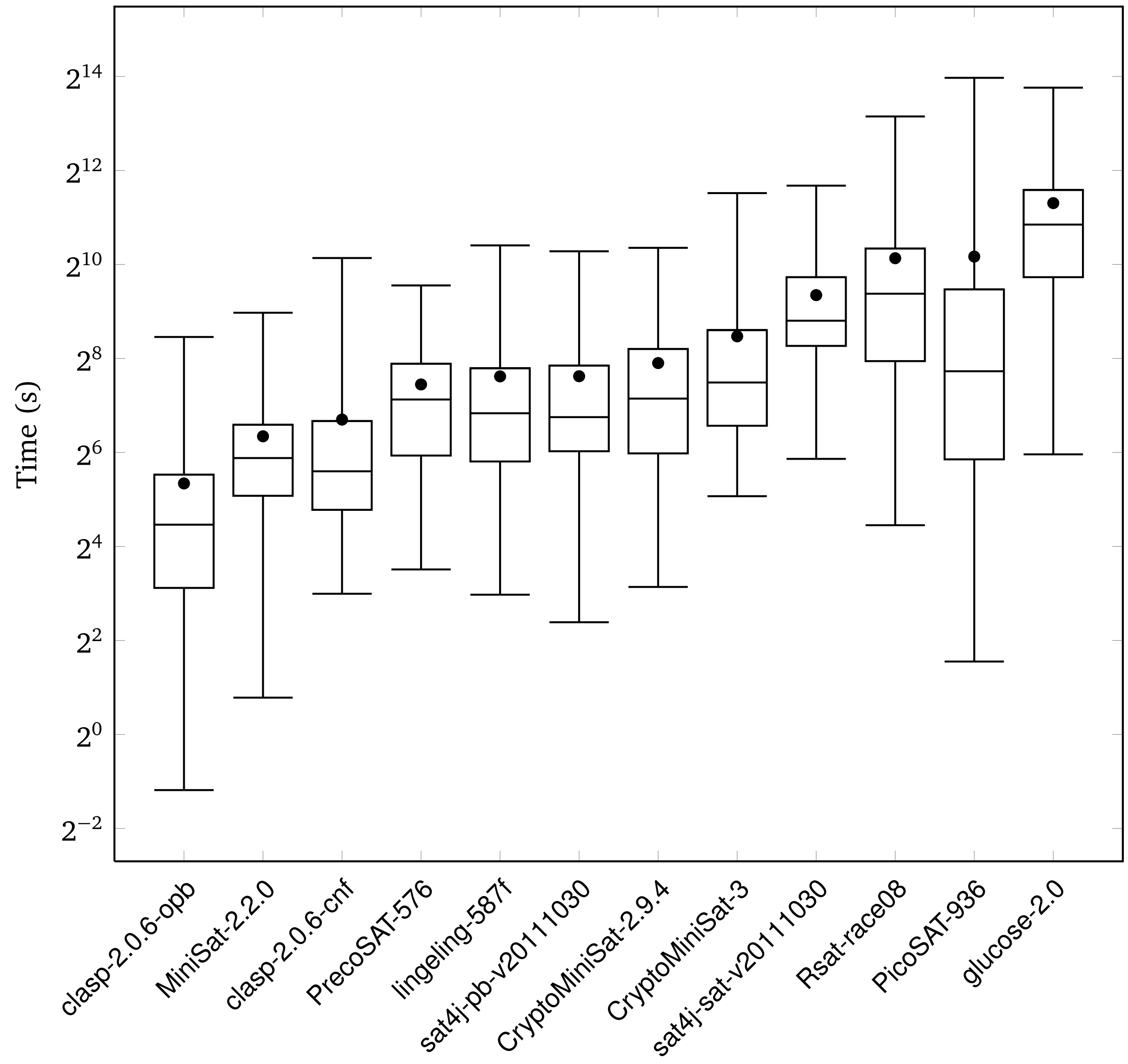


Figure 4.1: Box plot showing the quartiles of the sampling running time distributions for each of the 12 solvers we test. Dots indicate the mean running time. Each sample has the size $n = 100$.

other solvers (**MiniSat-2.2.0**) are all significant at the 5% level. We do not have enough evidence to rule out that **MiniSat-2.2.0** and **clasp-2.0.6-cnf** have the same mean, but the difference between **MiniSat-2.2.0** and **PrecoSat-576** is significant.

None of the difference between the solvers **PrecoSat-576**, **lingeling-587f**, **sat4j-pb-v20111030**, and **CryptoMiniSat-2.9.4** are significant. Although we measured the mean of **CryptoMiniSat-2.9.4** to be smaller than **CryptoMiniSat-3**, we do not have enough evidence to conclude that they are really different. Lastly, the differences between the worst solver, **glucose-2.0**, and all the other solvers were all found to be significant.

Table 4.1: All pairwise significance tests on the difference between the mean running times (the Games-Howell procedure) for the 12 solvers that we test. The solvers are ordered by the mean running time. Each sample has the size $n = 100$.

Solver 1	Mean (s)	Solver 2	Difference (s)	p-value
clasp-2.0.6-opb	40.48	**MiniSat-2.2.0**	40.66	**0.003**
		clasp-2.0.6-cnf	63.35	**0.025**
		PrecoSAT-576	133.84	**0.000**
		lingeling-587f	155.86	**0.000**
		sat4j-pb-v20111030	156.26	**0.000**
		CryptoMiniSat-2.9.4	198.43	**0.000**
		CryptoMiniSat-3	314.02	**0.000**
		sat4j-sat-v20111030	610.32	**0.000**
		Rsat-race08	1,081.14	**0.000**
		PicoSAT-936	1,108.28	**0.004**
		glucose-2.0	2,486.77	**0.000**
MiniSat-2.2.0	81.14	**clasp-2.0.6-cnf**	22.69	0.987
		PrecoSAT-576	93.18	**0.000**
		lingeling-587f	115.20	**0.001**
		sat4j-pb-v20111030	115.60	**0.000**
		CryptoMiniSat-2.9.4	157.77	**0.000**
		CryptoMiniSat-3	273.36	**0.000**
		sat4j-sat-v20111030	569.66	**0.000**
		Rsat-race08	1,040.48	**0.000**
		PicoSAT-936	1,067.62	**0.007**
		glucose-2.0	2,446.11	**0.000**
clasp-2.0.6-cnf	103.84	**PrecoSAT-576**	70.48	0.070
		lingeling-587f	92.51	0.084
		sat4j-pb-v20111030	92.91	0.051
		CryptoMiniSat-2.9.4	135.08	**0.003**
		CryptoMiniSat-3	250.66	**0.000**
		sat4j-sat-v20111030	546.97	**0.000**
		Rsat-race08	1,017.78	**0.000**
		PicoSAT-936	1,044.93	**0.010**
		glucose-2.0	2,423.41	**0.000**
PrecoSAT-576	174.32	**lingeling-587f**	22.03	1.000
		sat4j-pb-v20111030	22.43	0.999
		CryptoMiniSat-2.9.4	64.59	0.647
		CryptoMiniSat-3	180.18	**0.032**
		sat4j-sat-v20111030	476.49	**0.000**
		Rsat-race08	947.30	**0.000**
		PicoSAT-936	974.45	**0.022**
		glucose-2.0	2,352.93	**0.000**

Solver 1	Mean (s)	Solver 2	Difference (s)	p-value
lingeling-587f	196.35	**sat4j-pb-v20111030**	0.40	1.000
		CryptoMiniSat-2.9.4	42.57	0.992
		CryptoMiniSat-3	158.16	0.165
		sat4j-sat-v20111030	454.46	**0.000**
		Rsat-race08	925.28	**0.000**
		PicoSAT-936	952.42	**0.030**
		glucose-2.0	2,330.90	**0.000**
sat4j-pb-v20111030	196.75	**CryptoMiniSat-2.9.4**	42.17	0.990
		CryptoMiniSat-3	157.76	0.154
		sat4j-sat-v20111030	454.06	**0.000**
		Rsat-race08	924.88	**0.000**
		PicoSAT-936	952.02	**0.030**
		glucose-2.0	2,330.51	**0.000**
CryptoMiniSat-2.9.4	238.91	**CryptoMiniSat-3**	115.59	0.671
		sat4j-sat-v20111030	411.89	**0.000**
		Rsat-race08	882.71	**0.000**
		PicoSAT-936	909.85	**0.047**
		glucose-2.0	2,288.34	**0.000**
CryptoMiniSat-3	354.50	**sat4j-sat-v20111030**	296.31	**0.009**
		Rsat-race08	767.12	**0.000**
		PicoSAT-936	794.27	0.155
		glucose-2.0	2,172.75	**0.000**
sat4j-sat-v20111030	650.81	**Rsat-race08**	470.81	0.113
		PicoSAT-936	497.96	0.810
		glucose-2.0	1,876.44	**0.000**
Rsat-race08	1,121.62	**PicoSAT-936**	27.15	1.000
		glucose-2.0	1,405.63	**0.000**
PicoSAT-936	1,148.77	**glucose-2.0**	1,378.48	**0.009**

Discussion

We conclude that the most interesting solvers to use in further experiments are **clasp** and **MiniSat**. It also seems that the use of pseudo-boolean constraints greatly speeds up the solving process, as **clasp-2.0.6-opb** is significantly better than **clasp-2.0.6-cnf** and **sat4j-pb-v20111030** is significantly better than **sat4j-sat-v20111030**. Furthermore, **sat4j**, which is written in Java and routinely loses to highly optimised solvers like **lingeling** and **CryptoMiniSat** in SAT competitions, was not significantly different from these when using the version supporting pseudo-boolean constraints.

4.2 Running time distribution

After some preliminary analysis of the running time of some test runs, it seems that the running time of **MiniSat** is approximately log-normal distributed. In this experiment, we aim to determine whether our assumptions of log-normality for the running time holds, or if not, to what degree it is violated. This is important for the rest of our experiments since a significant violation could invalidate the results of other analyses (such as e.g. the computation of confidence intervals).

To test our assumption of lognormality, we can use a normality test such as the Shapiro-Wilk test on the log-transformed sample. Formally, we assume the null hypothesis H_0 : *the log-transformed sample comes from a normal distribution*, and we compute the probability of observing a value for the test statistic which is at least as extreme as the one observed. We will sometimes report the p-value for the test statistic. A low p-value (below 0.05 for a significance level of 5%) indicates that the log-transformed sample may not come from a normal distribution and we *reject* the null hypothesis; otherwise, we do not have enough evidence to prove that the log-transformed sample doesn't come from a non-normal distribution (note that this does not imply that we accept the null hypothesis). In this experiment, we will report both the Shapiro-Wilk test statistic W, which is a measure of how much the sample deviates from the normal distribution, and the corresponding p-value.

One of the most frequently used graphical tools for comparing distributions is the *Q-Q plot*. In our case, we plot the quantiles of our log-transformed sample vs. the theoretical quantiles of the standard normal distribution. If the log-transformed sample comes from the normal distribution (regardless of the parameters of the underlying distribution), the points will fall along a single line. For reference, a straight line going through the points corresponding to the 25% and 75% quantiles is also drawn.

We also compute and list *skewness* and *excess kurtosis*, two parameters for the shape of the distribution. Skewness is a measure of symmetry; negative skewness means that the distribution is left-skewed, positive skewness means that the distribution is right-skewed, and skewness 0 means that the distribution is symmetric. For normality of our log-transformed sample, we would like the skewness to be as close to 0 as possible. Excess kurtosis is a measure of the height and the sharpness of the peak relative to the normal distribution, and it ranges from -2 to ∞, where a value of -2 signifies that no central peak and no tails exist. The excess kurtosis of the normal distribution is 0, and we would like the excess kurtosis of our log-transformed sample to be as close to that value as possible.

In [Bard et al., 2007], the authors investigate the running time of SAT solvers for the problem of solving sparse systems of low-degree multivariate polynomials over $\mathbb{GF}(2)$ (the MQ problem). This problem is important in cryptography because many cryptographic primitives are specified in terms of such polynomials. They claim that the distribution of the sample they obtained (on the encoding of an unspecified cryptographic problem) is log-normal with excess kurtosis (of the log-transformed samples) ranging from -0.38 to 2.41. However, they used a timeout with the SAT solver, so the upper (right) tail is not completely visible.

We will run **MiniSat** on two different types of reduced instances. First, we obtain a sample of $n = 1000$ runs for 21 rounds and 160 fixed hash bits; then, we obtain a sample of $n = 100$ runs for 22 rounds and 128 fixed hash bits. The smaller number of runs for the second sample is due to time constraints.

Results

From the Q-Q plot in Figure 4.2, we see that the 21-round sample is not quite log-normal, since the right tail is shorter than it would be if the sample truly came from a log-normal distribution. This is also reflected in the sample skewness which was measured to be -0.594 (see Table 4.2) and the Shapiro-Wilk test which rejects the hypothesis that the sample came from a log-normal distribution.

The Q-Q plot shows that the 22-round sample is virtually indistinguishable from a sample from a true log-normal distribution, although the sample skewness was measured to be -0.287. Here we do not have enough evidence to reject the hypothesis that the sample came from a log-normal distribution.

Discussion

From these results we conclude that, although close, the running time distributions are not completely log-normal. Since the sample for the more difficult instance was closer to log-normal, we can hypothesise that the running times are *asymptotically* log-normal, although this is difficult to prove.

We can try to explain the approximate log-normality as follows: Log-normal distributions arise from multiplicative processes, i.e. in this case, the running time of a single run can be seen as the product of many (positive) factors. We could explain these factors as being the time to find the correct value for a single variable; since the instances have only very few solutions (all of which are far apart[1]) compared to the size of the whole search space, we see that solutions are characterised by *combinations* of the values of the variables rather than each variable tending towards having a particular value. In this interpretation, we see two ways to explain why the more difficult instances are closer to the true log-normal distribution: 1. with more rounds, the solutions are expected to be further apart; and 2. with more variables, the law of large numbers ensures that the product of the individual times converges faster towards log-normality.

Since the running time distribution for the easier instance is quite left-skewed, the approximate method of calculating a confidence interval for the mean of a log-normal distribution is going to overestimate the underlying mean. For this reason, we are better off using either the usual method of computing a confidence interval for the mean of a sample from the normal distribution or computing a boostrap confidence interval [Huber, 2012].

[1]The Hamming distance between the solutions is great. The Hamming distance is the number of positions in which two sequences of bits differ.

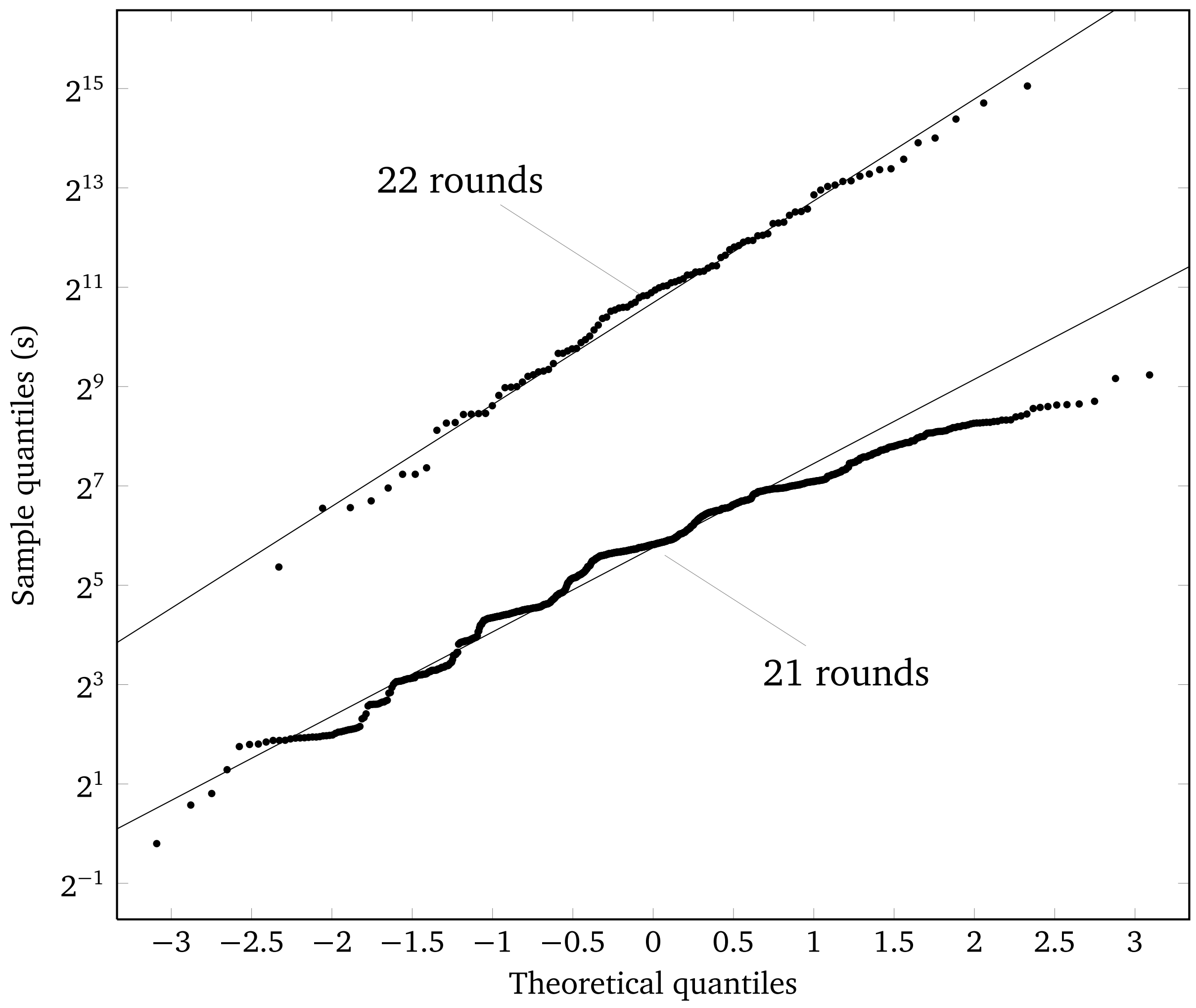


Figure 4.2: Q-Q plot showing the sample quantiles of the time it takes to find a 21-round preimage with 160 fixed hash bits and a 22-round preimage with 128 fixed hash bits using **MiniSat** vs. the theoretical quantiles of the standard normal distribution.

Table 4.2: The number of samples n, the Shapiro-Wilk test statistic W and p-value, skewness, and kurtosis for two different (log-transformed) samples.

Configuration	n	W	p	Skewness	Excess kurtosis
21 rounds, 160 fixed hash bits	1000	0.972	< 0.001	−0.594	0.146
22 rounds, 128 fixed hash bits	100	0.990	0.633	−0.287	−0.184

4.3 Reduced instances

Next, we investigate how running time varies as a function of the three difficulty parameters, number of rounds, number of fixed hash bits, and number of fixed message bits. Knowing this (1) makes it easier to estimate the difficulty of an instance and (2) may give some insights into the strengths and weaknesses of the both SHA-1 and the SAT approach. Regarding the first point, being able to estimate the difficulty of an instance is important since it allows us to design experiments that are expected to finish within a certain time limit. It may also allow us to extrapolate some results obtained on easier instances to more difficult instances.

For the number of rounds, we already know (through preliminary testing) that instances encoding 16 to 20 rounds are very easy (they are solved in less than a minute), 21 rounds is easy (1 to 2 minutes), 22 is hard (around 5 hours), and 23 is extremely hard (2 to 3 days). In order to see how even higher number of rounds affect the running time, we will reduce the difficulty by only fixing 16 (out of 160) bits of the hash. Similarly, in order to see how the number of fixed message bits and the number of hash bits affect the difficulty, we will reduce the difficulty by only solving for 21 (out of 80) rounds.

Results and discussion

Figure 4.3 shows the mean running time for every possible number of rounds from 16 to 80. There appear to be three distinct "phases": (1) between 16 and 21 rounds, the mean running time is less than $2^{-2.30}$ s; (2) between 21 and 27 rounds there is a very steep and almost perfectly exponential curve going from $2^{-2.30}$ s at 20 rounds to $2^{8.12}$ s at 27 rounds; (3) between 27 and 80 rounds, the mean running time barely grows from approximately $2^{8.12}$ to approximately $2^{9.61}$. It is important to stress the fact that this curve is limited to attacks where 16 hash bits are fixed, although we may have reason to suspect that the difficulty of the instance scales more or less exponentially with the number of fixed hash bits (see Figure 4.4).

We observe that the overall shape of the curve is quite different from the results obtained in [Rivest et al., 2008] for SAT-based analysis of the compression function of their proposed hash function MD6: "Moreover, after 6–7 rounds [out of at least 80], both running time and memory usage appear to grow superexponentially in the number of rounds." This could indicate either that their encoding of MD6 was poor or that MD6 is intrinsically harder than SHA-1.

One explanation for the rapid increase in mean running time starting for instances with 22 and more rounds might come from looking at the occurrences of message words in the message schedule (see Table 4.3). We see that 22 rounds is the lowest number of rounds where each message word occurs more than once in the full formula; variables encoding words that only appear once in the formula are essentially just extra degrees of freedom. It could also simply be the fact that the final hash value (which is completely fixed in these experiments) is also the state of the last 5 rounds. In other words, for n rounds, the state for rounds $n-5$ to $n-1$ is known.

Figure 4.4 shows the mean running time as a function of the number of fixed hash bits. Due to time constraints, we were not able to obtain samples for more than 96

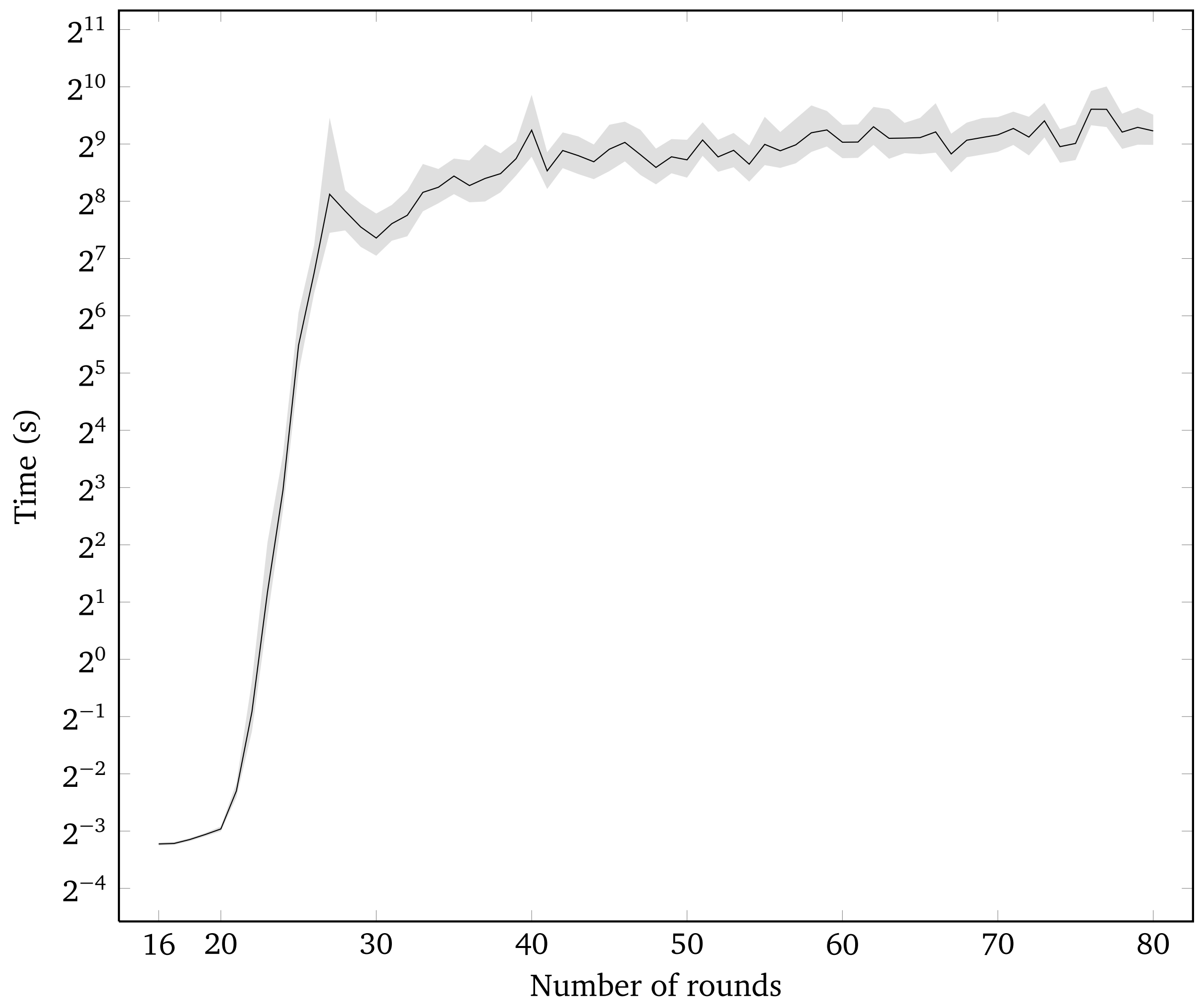


Figure 4.3: The mean time (with uncorrected 95% confidence band) to find a preimage for 16 fixed hash bits as a function of the number of rounds.

fixed hash bits for 22 rounds. As stated in section 2.4, we expected that the running time would be exponential in the number of fixed hash bits. However, disregarding the instances with fewer than 16 fixed hash bits (since they are very easy to solve and the running time could easily be dominated e.g. by time it takes to parse the instance file), it appears that the mean running time for both 21 and 22 rounds is sub-exponential.

It appears that the difficulty increases faster for a lower number of fixed hash bits (e.g. for less than 48 bits for 22 rounds) and slower for a higher number (e.g. for more than 64 bits for 22 rounds). Figure 4.5 shows the mean running time as a function of the number of fixed message bits. The first thing to notice is that both fixing a low number of message bits and fixing a high number of message bits yield instances that are easier than if we fix a number lying in-between. This is contrary to our prediction that the difficulty will simply be exponential in the number of unfixed bits; however, one explanation for this trend could be that fixing many message bits lowers the upper bound on the size of the search space (2^n, where n is the number of independent variables), but it also lowers the expected number of solutions (2^{n-160}).

While certainly not a rigorous method, extrapolating the trends seen in Figure 4.5

Table 4.3: The number of occurrences of each message words in the message schedule of round-reduced SHA-1 instances. For 16 rounds, no word appears more than once, and for 22–80 rounds, every word appears more than once. (Note that each word additionally appears exactly once in rounds 0 to 15.)

Rounds	$\mathbf{W}_0$	$\mathbf{W}_1$	$\mathbf{W}_2$	$\mathbf{W}_3$	$\mathbf{W}_4$	$\mathbf{W}_5$	$\mathbf{W}_6$	$\mathbf{W}_7$	$\mathbf{W}_8$	$\mathbf{W}_9$	$\mathbf{W}_{10}$	$\mathbf{W}_{11}$	$\mathbf{W}_{12}$	$\mathbf{W}_{13}$	$\mathbf{W}_{14}$	$\mathbf{W}_{15}$
16	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
17	1	·	1	·	·	·	·	·	1	·	·	·	·	1	·	·
18	1	1	1	1	·	·	·	·	1	1	·	·	·	1	1	·
19	1	1	2	1	1	·	·	·	1	1	1	·	·	1	1	1
20	2	1	3	2	1	1	·	·	2	1	1	1	·	2	1	1
21	2	2	3	3	2	1	1	·	2	2	1	1	1	2	2	1
22	2	2	4	3	3	2	1	1	2	2	2	1	1	3	2	2

using straight lines yields an intersection at the point $\left(389.11, 2^{50.60}\right)$. While the straight-line extrapolations are not supported by hard data (and thus also not the intersection point), the location of the intersection point is interesting because it is close to the point where we expect (on average) to find only a single solution. As we already explained and saw in Table 4.3, for 21 rounds, the 32 bits of w_7 are essentially just extra degrees of freedom, i.e. the point where we expect to find only a single solution is at $512 - 160 + 32 = 384$ fixed message bits (for 21 rounds). Thus it seems that the SAT solver benefits from having more solutions in the search space, even though the whole search space is bigger.

As an overall conclusion, we suggest that it is better to reduce the difficulty of instances by fixing hash bits rather than using fewer rounds or fixing fewer message bits, since this appears to be the function showing the simplest relationship between the parameter and the mean running time.

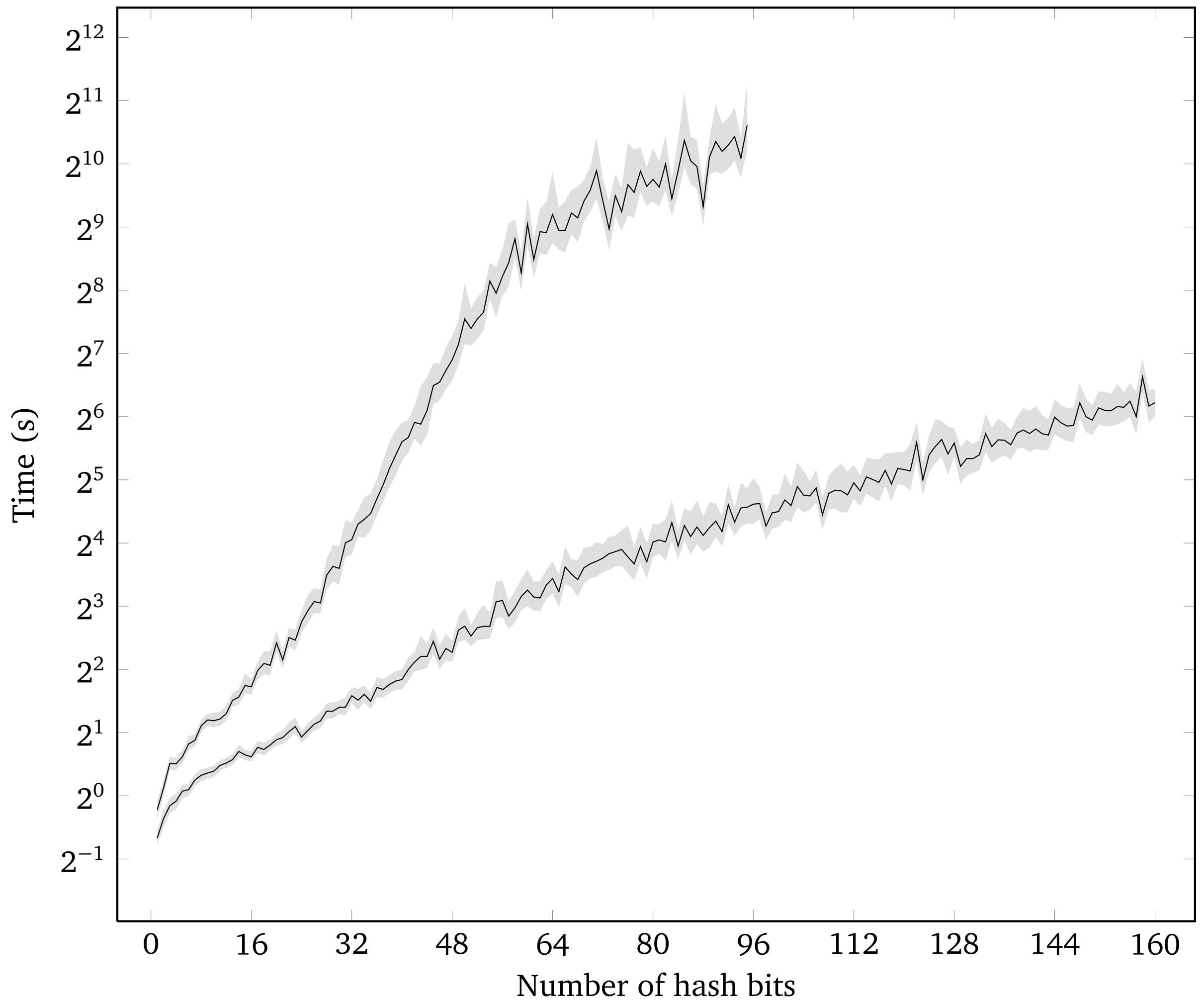


Figure 4.4: The mean time (with uncorrected 95% confidence band) to find 21- and 22-round preimages as a function of the number of fixed hash bits.

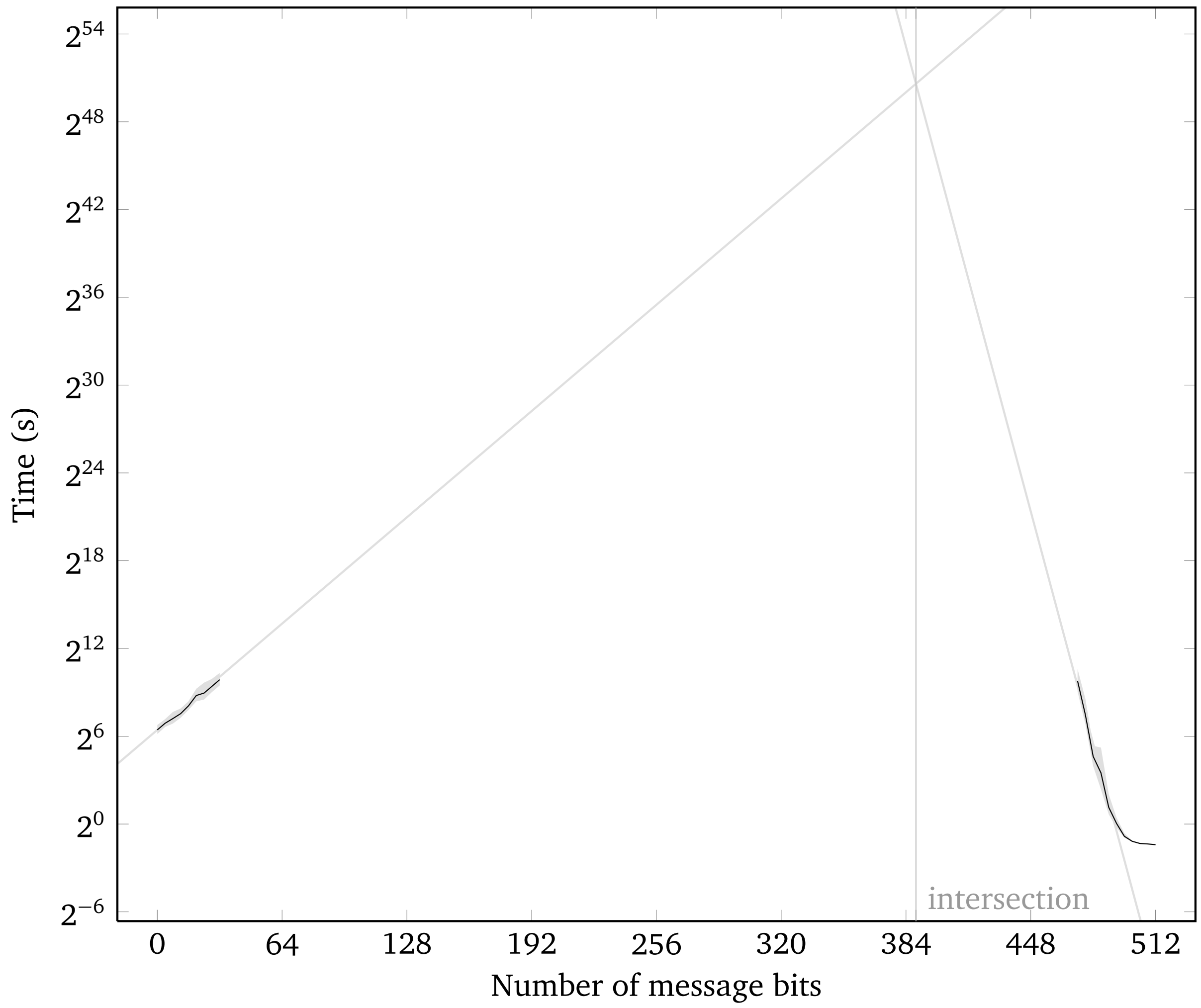


Figure 4.5: The mean time (with uncorrected 95% confidence band) to find 21-round preimages as a function of the number of fixed message bits.

4.4 Fixing specific message/hash bits

In the previous experiment, we investigated what effect fixing a certain number of message and hash bits has on the mean running time for the SAT solver. However, the choice of exactly which bits to fix was made at random for each instance. Because SHA-1 was designed to be a hash function, we expect each message and hash bit to contribute equally to the difficulty of the instance. This does not necessarily hold for reduced-round versions, however. Our goal in this experiment is, therefore, to determine whether there exist certain bits which, when fixed, change the average difficulty of the problem in one direction or the other.

We could form $512+160$ hypotheses of the form "the running time when fixing bit i is the same as the running time when not fixing bit i". It would take too long to obtain all the samples, however. Instead, we generate a larger number of instances where both message and hash bits are fixed at random. Since we know for each instance exactly which bits were fixed, we can, for each bit, divide the sample in two: the runs where the bit was fixed and the runs where the bit was unfixed. Since the other bits were fixed at random, their effects cancel out with a large enough sample, and we obtain a mean difference for each bit.

We generated and solved 2370 instances, each with 21 rounds, 32 fixed message bits, and 96 fixed hash bits. On average, we expect each message bit to have $32 \cdot 2370/512 = 148.125$ samples for the case when they are fixed and $(512-32) \cdot 2370/512 = 2221.875$ samples for the case when they are not fixed. Similarly, we expect each hash bit to have $96 \cdot 2370/160 = 1422$ samples for the case when they are fixed and $(160-96) \cdot 2370/160 = 948$ samples for the case when they are not fixed.

Results

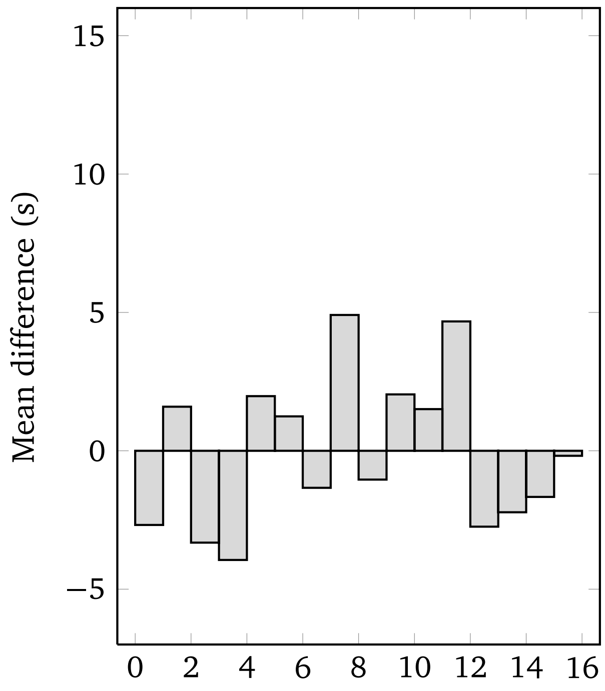
Although we have collected data for each variable individually, we will combine and describe the results from two perspectives. Recall that the message consists of 16 words of 32 bits each. Likewise, the hash consists of 5 words of 32 bits each.

Figure 4.6 shows the combined results of each bit and each word separately; i.e. for each message/hash word, we compute the average effect of fixing each bit within that word; similarly, for each bit number, we compute the average effect of fixing each bit at that position across all the message/hash words. For example, according to Figure 4.6c, fixing a random bit of hash word 0 will on average increase the running time of the SAT solver by nearly 6 seconds.

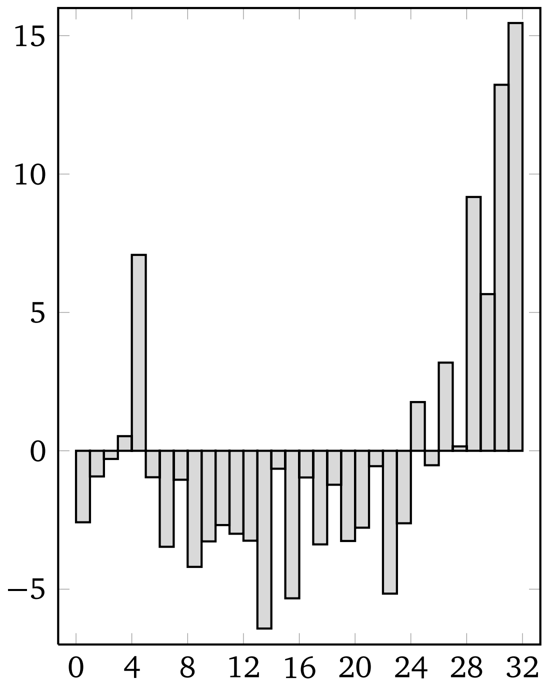
Discussion

Firstly, there is a very clear pattern for the hash word number: fixing bits in words 0 and 1 increases the difficulty of the instance, while fixing bits in words 2 to 4 decreases the difficulty.

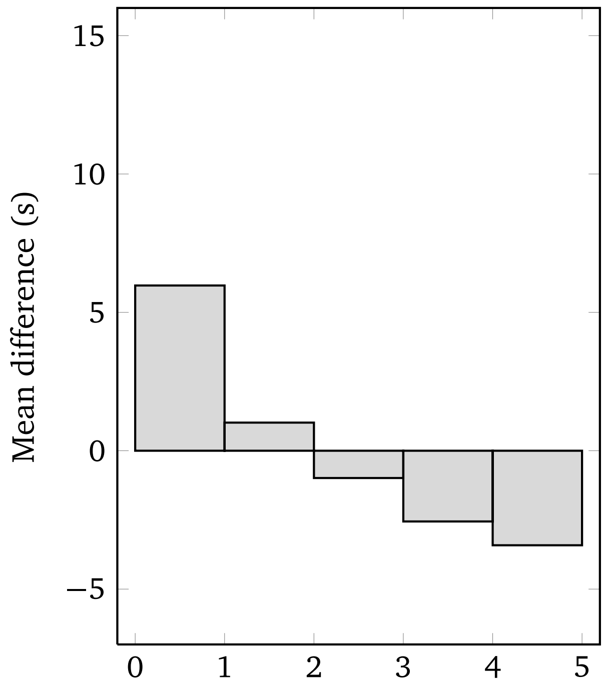
Secondly, there is also a fairly clear pattern for the bits in each word of both the message and the hash: fixing the middle bits decreases the difficulty while fixing the most significant bits (and one or two of the least significant bits) increases the difficulty drastically.



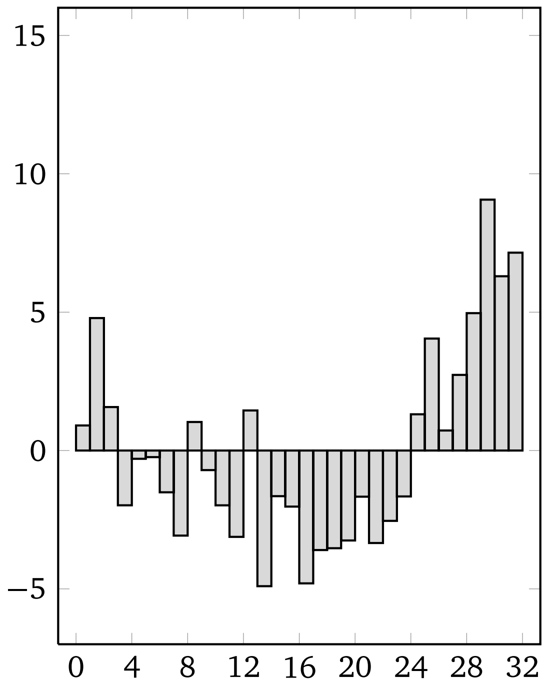
(a) Message word number



(b) Message word bit



(c) Hash word number



(d) Hash word bit

Figure 4.6: The effects on the mean running time when certain message/hash bits are fixed.

For fixing bits in the message words, there doesn't immediately seem to be a clear pattern. However, on closer inspection, there seems to be some correlation with the number of occurrences of that word in the message schedule for 21 rounds (see Table 4.3). For example, notice the peaks at $\mathbf{W}_3$ and $\mathbf{W}_4$ where there is a decrease the running time; those are the two most constrained words in the message schedule for 21 rounds. The least constrained word, $\mathbf{W}_7$, is also where fixing bits seems to increase the running time the most.

4.5 Encodings

In subsection 2.2.2 and subsection 2.2.3 we described two CNF encodings of modular addition: standard Tseitin-encoded ripple-carry adder circuits and **Espresso**-minimised (and CNF-encoded) pseudo-boolean constraints; we label them **Tseitin** and **Espresso**, respectively. In [Morawiecki and Srebrny, 2010] the authors presented their toolkit, **CryptLogVer**, which generates a CNF formula from a circuit specified in the Verilog hardware description language (HDL). The toolkit relies on proprietary software from Altera[2] to create the circuit. Since Verilog looks more like a usual programming language, adapting the SHA-1 algorithm from C is straightforward, and the authors even include Verilog code for SHA-1 as one example of an algorithm that can be encoded in SAT using their toolkit.

The purpose of the experiment is twofold: (1) to find out whether adders are better encoded using **Espresso**-minimised pseudo-boolean constraints or using the Tseitin transformation; and (2) to find out whether the **CryptLogVer** toolkit can compete with either of the two previous encodings.

We wanted to run the experiment for 21 rounds and 160 hash bits, 22 rounds and 48 hash bits, and 80 rounds and 4 hash bits, but due to an unexpected error (or perhaps feature of) the **CryptLogVer** encoding, we must always fix all the message bits. If we don't, we will sometimes find solutions where the hash found is not the correct hash for the message found (i.e. the encoding is not sound). For this reason, we were only able to obtain complete samples for 21 rounds.

We use **MiniSat-2.2.0** with default settings for all three encodings.

Results

Our results are outlined in Table 4.4 and Table 4.5. The **Espresso** encoding for the adders outperforms the **Tseitin** and the **CryptLogVer** encodings by a factor of approximately 2.48. It is also the encoding with the fewest variables and the highest number of clauses. The **CryptLogVer** encoding is significantly better than the **Tseitin** encoding (for the adders), but the two methods appear to have a similar ratio of clauses to variables (4.88 and 4.85, respectively), while the **Espresso** encoding has a ratio of 30.32.

[2]Vendor of digital circuits, including FPGA devices and ASICs.

Table 4.4: The number of variables, number of clauses, their ratio, and the mean running time (with uncorrected 95% confidence intervals) for the solver for three different encodings; two imperative handcrafted encodings where the only difference is the encoding of the adders (**Espresso** and **Tseitin**), and one encoding where we used the **CryptLogVer** toolkit. Each sample has the size $n = 100$.

Encoding	Rounds	Variables	Clauses	Ratio	Mean (s)	Mean 95% CI (s)	
espresso	21	3,968	120,328	30.32	76.04	63.92	92.64
	22	4,128	126,420	30.62	–	–	–
	80	13,408	478,476	35.69	–	–	–
CryptLogVer	21	9,322	45,494	4.88	188.77	156.35	232.88
	22	9,792	48,147	4.92	–	–	–
	80	44,812	248,220	5.54	–	–	–
Tseitin	21	17,404	84,411	4.85	272.80	228.76	329.95
	22	18,060	86,791	4.81	–	–	–
	80	56,108	223,551	3.98	–	–	–

Table 4.5: All pairwise significance tests on the difference between the mean running times (the Games-Howell procedure) for the 3 encodings that we test. The encodings are ordered by mean running time. Each sample has the size $n = 100$.

Encoding 1	Mean (s)	Encoding 2	Difference (s)	p-value
espresso	76.04	**CryptLogVer**	112.73	**0.000**
		Tseitin	196.75	**0.000**
CryptLogVer	188.77	**Tseitin**	84.03	**0.026**

Discussion

Firstly, we remark that the number of variables and number of clauses that we obtained for the **Tseitin** encoding are very similar to those obtained for 80-round SHA-1 in [Srebrny et al., 2007]: "We obtained a propositional formula ... with nearly 55 thousand propositional variables and nearly 235 thousand clauses." (By comparison, we get 56,108 variables and 223,551 clauses; the differences can probably be attributed to other parts of SHA-1 that were *not* encoded using the Tseitin transformation, such as the round-dependent logical functions.)

The large difference in the number of variables and clauses between the **Espresso** and the **Tseitin** encodings indicate that a large part of the encoding comes from the adders alone and that there is a great potential both for making the instance more compact (in terms of the number of variables) and for making the instance easier to handle for the solver.

We can also safely conclude that adders encoded using **Espresso**-minimised pseudo-boolean constraints are better for the solver, even though it requires a lot more clauses than the Tseitin-encoded counterpart. In fact, we might suspect that the **CryptLogVer** toolkit also uses a variant of the Tseitin encoding, since the two encodings have very similar ratios of clauses to variables. Note, however, that the instance we obtain using the **CryptLogVer** toolkit is much smaller than our naïve Tseitin encoding. We suspect that this is due to the use of minimisation/simplification algorithms in **CryptLogVer**.

We guess that the fewer variables and greater number of clauses help the solver because the encoding has better implicativity, i.e. it is closer to arc-consistency than the other encodings. Also, longer clauses are in general better than shorter clauses because "propagation in a SAT solver is roughly proportional to the number of clauses, independent of their size" [Eén and Biere, 2005].

Unless otherwise specified, we will be using **Espresso**-minimised pseudo-boolean constraints as the encoding in all further experiments.

4.6 Pseudo-boolean and unary/binary constraints

In section 4.1 we compared the performance of both plain CNF solvers and OPB solvers and found **clasp-2.0.6** and **MiniSat** to be the best solvers (and that in both cases where we had a solver that accepted both CNF and OPB input, the OPB instance was significantly easier to solve). In section 4.5 we compared the performance of different CNF encodings of 32-bit modular addition and found **Espresso**-minimised pseudo-boolean constraints to be significantly easier than any other CNF encoding.

As a further step, we now compare the best CNF encoding (**Espresso**-minimised pseudo-boolean constraints) on the best CNF solver (**MiniSat**) with the OPB encoding (using pseudo-boolean constraints) on the best OPB solver (**clasp**) and with an encoding using explicit unary/binary constraints on the best CNF solver (**MiniSat**) modified to support unary/binary constraints. In order to evaluate scalability as well, we will test each configuration on a range of difficulties.

We defined unary/binary constraints in subsection 2.2.4. We added support for reasoning directly with these constraints to **MiniSat** by extending the input language with an "h" line so that e.g. the constraint $x_1 + x_2 + x_3 = x_4 \circ x_5$ can be expressed using the line

```
h 1 2 3 0 4 5 0.
```

We also extend the input language with a "d" line so that we can indicate to the solver that certain variables are not to be branched on; in particular, the dummy variables mentioned in subsection 2.2.4 are needed only to satisfy the syntactic requirements of the constraint. For example, the line

```
d 6 -7 0
```

would indicate that variable 6 is a potential decision variable, while variable 7 is not.

Whenever a value is assigned to a variable that appears in the constraint, we check the number of ways in which each variable in the constraint can still be assigned. To check all the ways in which the clause can be satisfied given the current partial valuation, we first calculate a lower and an upper bound for the value of the left-hand

$c_1 + x_2 + y_2 + z_2$	d_0	c_3	w_2
~~0~~	0	0	0
~~1~~	0	0	1
2	0	1	0
3	0	1	1
4	~~1~~	0	0
~~5~~	~~1~~	0	1
~~6~~	~~1~~	1	0
~~7~~	~~1~~	1	1

Figure 4.7: All possible assignments to the RHS of the unary/binary constraint $c_1 + x_2 + y_2 + z_2 = d_0 \circ c_3 \circ w_2$ under the valuation v where $v(c_1) = 1$, $v(x_2) = 1$, and $v(d_0) = 0$. Impossible assignments have been struck out. We see that we can propagate $v(c_3) = 1$ because no other value could satisfy the constraint.

side (LHS). The lower bound is the number of satisfied literals on the LHS, while the upper bound is the number of satisfied literals on the LHS plus the number of undefined literals on the LHS. For example, given the constraint

$$c_1 + x_2 + y_2 + z_2 = d_0 \circ c_3 \circ w_2$$

and a partial valuation v where $v(c_1) = 1$, $v(x_2) = 1$, and $v(d_0) = 0$, the value of the LHS is bounded by the interval $[2, 4]$ (see Figure 4.7).

Once we have established the bounds for the LHS, we iterate over all the assignments to literals in the RHS which would satisfy the LHS. Potential assignments to the RHS which are incompatible with v are skipped. In this case, since $v(d_0) = 0$, we can immediately rule out 4 as a possible value of the RHS. For each variable in the RHS, we check whether both 0 and 1 occur; if not, we propagate the variable. In our example, w_2 could be either 0 or 1, so we have no valid reason why it should be one or the other. However, c_3 can only be 1, so we can propagate $v(c_3) = 1$.

In order to facilitate conflict analysis, we always have to provide a reason for why a variable was propagated. We simply take the conjunction of the literals in the constraint which are currently defined as the condition and the implied variable(s) as the consequence. In our example, we would derive the fact $(c_1 \wedge x_2 \wedge d_0) \to c_3$, which can be written in clausal form as $\neg c_1 \vee \neg x_2 \vee \neg d_0 \vee c_3$. If a conflict is detected, we similarly create a reason clause by negating the combination of the variables in the constraint which are currently defined. Since these “reason clauses” are only needed for conflict analysis, we never add them to the clause database, but free them as soon as they are no longer needed, i.e. because we backtracked past the decision level where the propagation was made.

There are two known possible drawbacks to this direct implementation of unary/binary constraints. Firstly, we do not have a watchlist scheme for the constraint, so we must always visit every unary/binary constraint that contains the variable x whenever a value is assigned to x. Secondly, the generated reason clauses are not necessarily minimal.

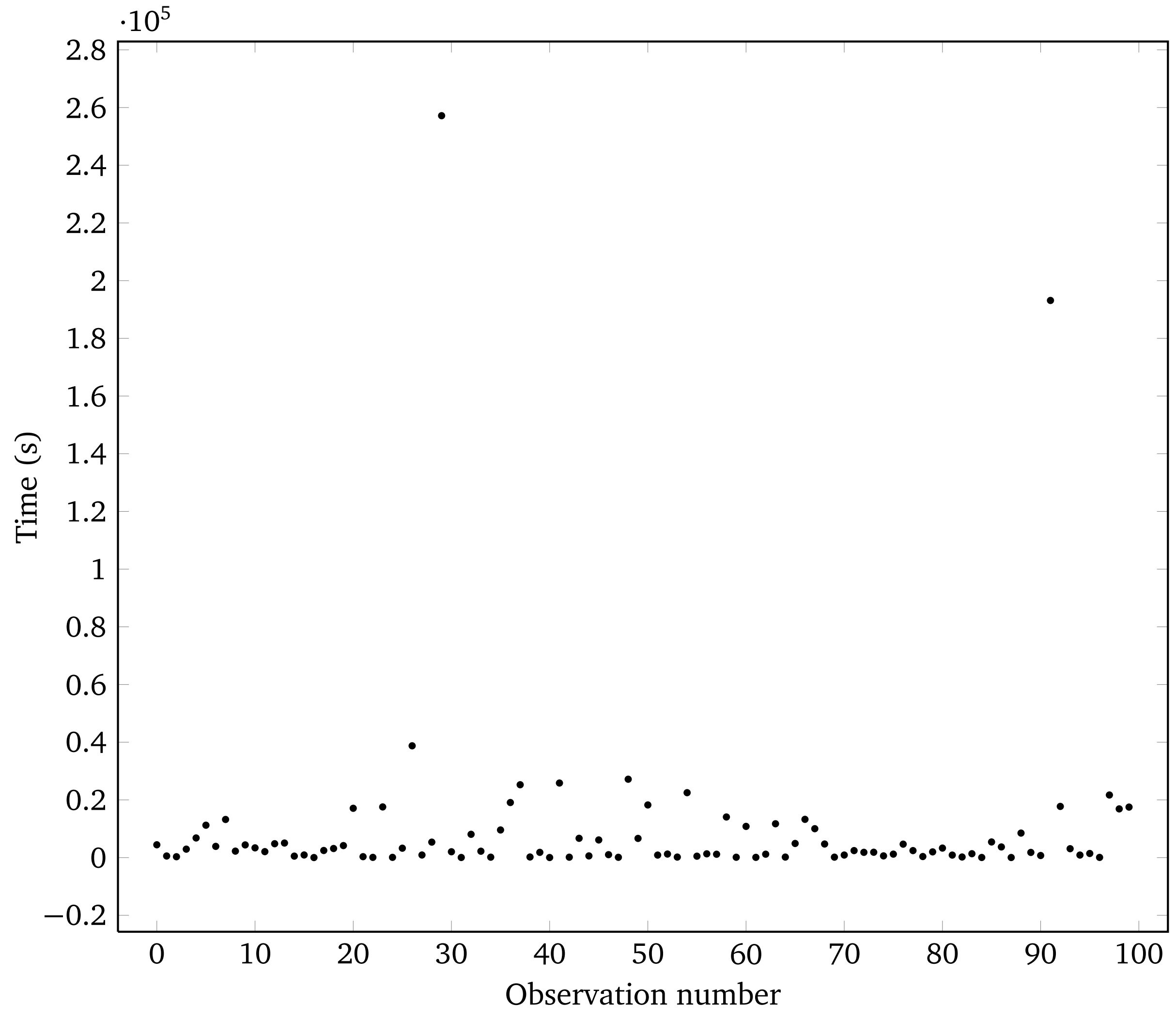


Figure 4.8: All observations (sorted) using **clasp** for 22 rounds and 128 fixed hash bits. Clearly, there are two outliers (observations 29 and 91). Without these two very influential observations, the mean time to solve drops from 9939.66 s to 5547.54 s.

Results

We first note that there are several apparent outliers in the data we obtained for **clasp**. An observation is considered an outlier if it lies outside the range $[Q_1 - 10(Q_3 - Q_1), Q_3 + 10(Q_3 - Q_1)]$, where Q_1 and Q_3 are the lower and upper quartiles, respectively [AST, 2008]. For example, two outliers are clearly visible in Figure 4.8, which shows all observations for 22 rounds and 128 fixed hash bits. We also find one such outlier in each of the samples obtained using **clasp** for 64 and 96 fixed hash bits. There are no outliers in the other samples.

The mean times to find 22-round preimages for the different solvers/constraints and difficulties are shown in Figure 4.9. We also include **clasp-opb** with outliers removed. The general trend is as follows: for the easier instances, **clasp** is the fastest and **MiniSat** is the slowest, while for the harder instances, **clasp** (including the outliers) is the slowest and **MiniSat** is the fastest.

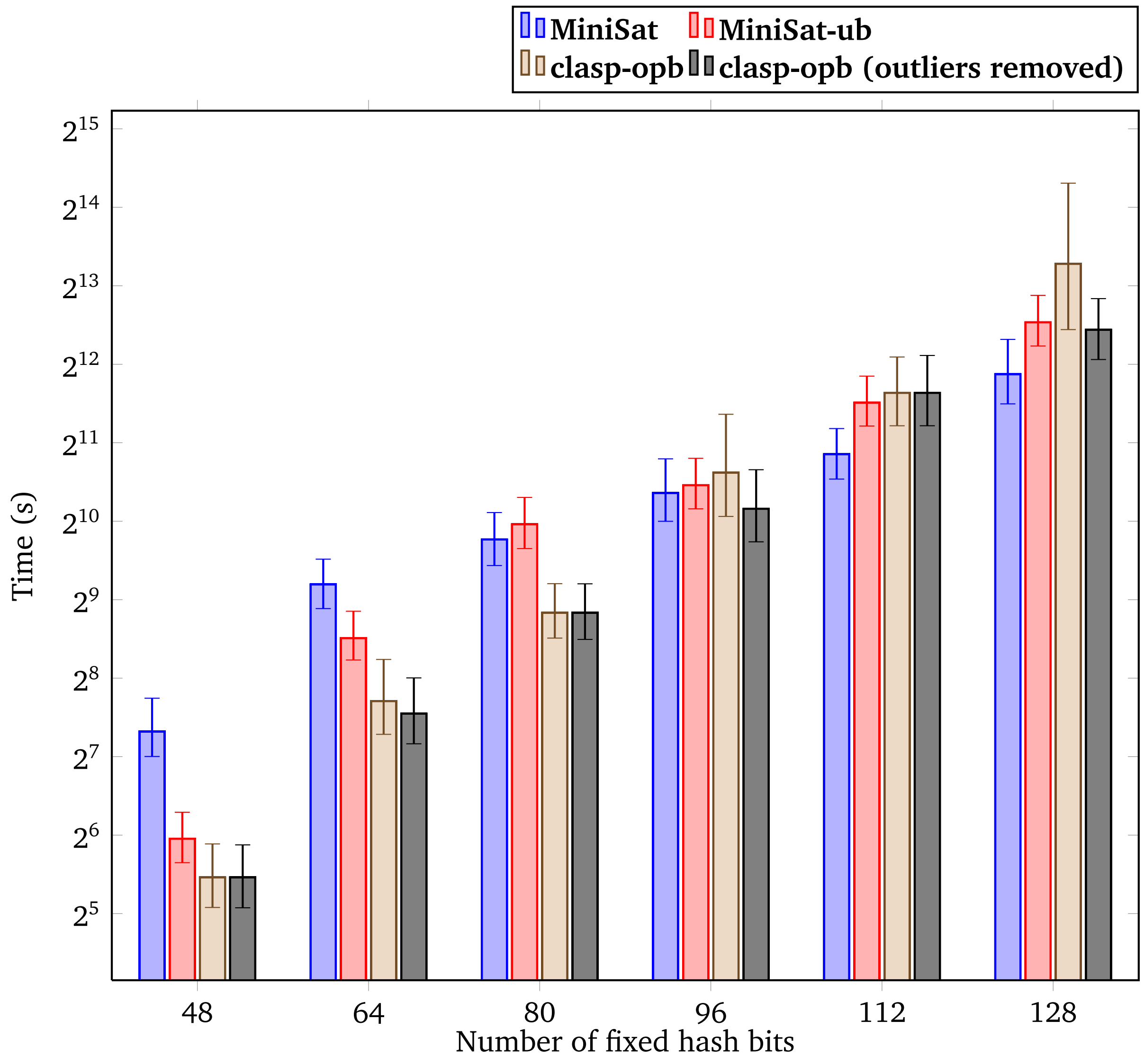


Figure 4.9: The mean time (with uncorrected 95% confidence intervals) to find a 22-round preimage with three solvers as a function of the number of fixed hash bits.

We test all pairwise differences between the solvers for significance; the results are listed in Table 4.6. For the easiest instance, with 48 fixed hash bits, **clasp-opb** and **MiniSat-ub** are both significantly better than **MiniSat**; however, we do not have enough evidence to conclude that there really is difference between **clasp-opb** and **MiniSat-ub**. For 96 fixed hash bits, there are no significant differences between any of the solvers. For 112 fixed hash bits, where there were no outliers, we find that **MiniSat** is significantly better than **clasp-opb**, however, for 128 fixed hash bits, even though the difference between the means is very large, we do not have enough evidence to say that they are really different (this is most likely a consequence of the outliers, which also affect the variability of the estimate of the mean).

Table 4.6: All pairwise significance tests on the difference between the mean running times (the Games-Howell procedure) for the 3 configurations that we test. Each sample has the size $n = 100$.

Hash bits	Configuration 1	Mean (s)	Configuration 2	Difference (s)	p-value
48	**clasp-opb**	44.06	**minisat-ub**	17.91	0.148
			minisat	115.72	**0.000**
	minisat-ub	61.96	**minisat**	97.81	**0.000**
64	**clasp-opb**	209.03	**minisat-ub**	155.48	**0.012**
			minisat	376.73	**0.000**
	minisat-ub	364.50	**minisat**	221.25	**0.013**
80	**clasp-opb**	456.16	**minisat**	414.38	**0.002**
			minisat-ub	539.77	**0.000**
	minisat	870.54	**minisat-ub**	125.38	0.701
96	**minisat**	1,312.36	**minisat-ub**	94.31	0.921
			clasp-opb	260.28	0.796
	minisat-ub	1,406.67	**clasp-opb**	165.96	0.908
112	**minisat**	1,849.47	**minisat-ub**	1,065.20	**0.017**
			clasp-opb	1,328.68	**0.041**
	minisat-ub	2,914.67	**clasp-opb**	263.49	0.898
128	**minisat**	3,748.15	**minisat-ub**	2,179.36	**0.036**
			clasp-opb	6,191.51	0.143
	minisat-ub	5,927.51	**clasp-opb**	4,012.15	0.443

Discussion

It seems that we can draw the following conclusions: Although slower than the other solvers/encodings for the easiest instances in this experiment, **MiniSat** scales well and is the fastest solver for the most difficult instances. The solver with support for unary/binary constraints, **MiniSat-ub**, seems to be better than **clasp-opb**, even for the more difficult instances, but this could depend a little bit on the cause of the outliers we observed with **clasp-opb**.

We have two possible explanations for why the plain CNF solver seems to scale better with more difficult instances: (1) pseudo-boolean and unary/binary constraints reduce the overall number of constraints (and literals) that the solver needs to keep track of, and so gives the solver a boost by giving it a smaller cache footprint. It is possible that this benefit is lost for the more difficult instances, since the solvers need to learn many more clauses which saturate the cache anyway; (2) unary/binary and pseudo-boolean constraints have no watched literal scheme and a more elaborate watched literal scheme, respectively.

One possible explanation for the outliers in **clasp** comes from Marijn Heule (through private correspondance with Mate Soos): **clasp** sets an absolute upper limit on the number of learnt clauses to three times the number of clauses and constraints in the original instance. Combined with restarts, this actually makes the solver incomplete, since it is not guaranteed to always make progress [Moskewicz et al., 2001]. Although unlikely to get stuck in an infinite loop, it could very well be possible that the progress slows down dramatically when the solver reaches the limit; by restricting the number of learnt clauses, the reasoning power also decreases (indeed, we can see the limit as a restriction on the resolution proof constructed by the solver). This would also explain why more difficult instances appear to have more outliers; easier instances also have less variation in the running time, so it is less likely that we get a value so extreme that the number of learnt clauses reaches the limit.

It would be interesting to investigate whether lowering the limit on the number of learnt clauses also lowers the threshold for when the time to solve an instance becomes much longer. If so, we can conclude that having an absolute upper limit on the number of learnt clauses is detrimental to the mean time to solve an instance.

4.7 XOR constraints and Gaussian elimination

In [Soos et al., 2009] and [Soos, 2010], the authors add support for XOR clauses to **MiniSat**. Explicit XOR constraints can improve the performance of the solver primarily in the following two ways. Firstly, an XOR constraint of n literals must be encoded using 2^{n-1} CNF clauses, all having n literals, if no new variables are introduced. The two-watch literal scheme [Moskewicz et al., 2001] also works for XOR clauses. This saves both memory and time. Secondly, sets of XOR clauses can be viewed as a set of linear equations over $\mathbf{GF}(2)$, and one can thus use other reasoning procedures such as Gaussian elimination to detect conflicts and propagations which were not detected using unit propagation alone.

Since XOR operations are ubiquitous in cryptography, the resulting solver supporting XOR clauses and Gaussian elimination was appropriately called **CryptoMiniSat**. Indeed, we have good reason to believe that XOR reasoning can speed up the solving of SHA-1, since the whole SHA-1 message schedule can be expressed as a 2560×512 matrix over $\mathbf{GF}(2)$.

To see whether Gaussian elimination speeds up the solver for SHA-1, we run **CryptoMiniSat** on the encoding using XOR clauses for the message schedule. Since the matrix of XOR constraints encoding the message schedule is bigger when solving for more rounds, we only try instances with 80 rounds. In order to ensure that Gaussian elimination was used at every decision level and that all possible matrices were used, we passed the options `--gaussuntil=15000 --maxnummatrixes=10 --maxmatrixrows=4096` to the solver.

Results

The Gaussian elimination procedure was called an average of 167,882 times per instance. Out of these, 0.84% resulted in a conflict and 7.75% resulted in a propagation.

From Table 4.7, we see that the mean running time appears to be lower when Gaussian elimination is enabled; however the confidence intervals overlap greatly.

Table 4.7: Running times for **CryptoMiniSat** when Gaussian elimination is enabled and disabled. Each sample has the size $n = 100$.

Configuration	Mean (s)	Mean 95% CI (s)	
CryptoMiniSat (Gaussian elimination)	861.68	613.78	1,349.66
CryptoMiniSat	1,049.77	815.80	1,389.95

Indeed, the p-value for the Games-Howell procedure is 0.394, which means that we do not have enough evidence to reject the hypothesis that there is no difference between the two means.

Discussion

We see that Gaussian elimination does help in *some* way: it is able to learn some facts which weren't discovered using unit propagation alone. However, it is questionable whether this is enough to help improve the running time. Of course, we have purposefully used a cut-off value which is very large, so it could be the case that the procedure runs more often than necessary.

It is hard to draw any conclusions about the mean running time when the observed difference is not statistically significant.

4.8 Preprocessing and simplification

In the influential paper [Eén and Biere, 2005], the authors described a preprocessing technique called *variable elimination*. The technique was first implemented as a standalone preprocessor called **SatELite** and later included in **MiniSat** under the **elim** option.

The purpose of preprocessing is to overcome naïve encodings that include "meaningless internal variables, equivalent literals, and unpropagated shallow facts". The authors state that encoding and preprocessing are "two sides of the same coin", i.e. that preprocessing is a general approach to optimising the encoding of an instance, and that preprocessing is unlikely to show large improvements for encodings that are already highly optimised.

MiniSat includes two other simplification options: **asymm** and **rcheck**, enabling *asymmetric branching* and checking whether added clauses are already implied, respectively. Both techniques work by propagating, for each clause, the negations of its literals. Asymmetric branching propagates all except (the negation of) one literal, and if a conflict is detected, the literal can be removed. The other option, **rcheck**, propagates (the negation of) each literal of the clause in turn, and if a literal is found to already have been implied, we know that the clause is redundant and can be removed from the instance.

In this experiment, we aim to determine how much these preprocessing and simplification options affect the running time of the SAT solver. We will try five different

Table 4.8: Mean running times (with uncorrected 95% confidence intervals) and all pairwise significance tests on their differences (the Games-Howell procedure) for the 5 combinations of options that we test. The configurations are ordered by the mean running time. Each sample has the size $n = 100$.

Configuration 1	Mean (s)	Mean 95% CI (s)		Conf. 2	Difference (s)	p-value
(all)	78.28	64.58	99.20	**(none)**	5.11	0.994
				elim	10.80	0.899
				asymm	14.28	0.768
				rcheck	35.34	0.091
(none)	83.38	69.55	106.21	**elim**	5.70	0.991
				asymm	9.18	0.947
				rcheck	30.23	0.214
elim	89.08	74.25	107.65	**asymm**	3.48	0.999
				rcheck	24.54	0.412
asymm	92.56	76.65	111.87	**rcheck**	21.05	0.575
rcheck	113.61	95.13	139.55			

combinations of options and compare them all against each other: (1) no option; (2) only **elim**; (3) only **async**; (4) only **rcheck**; and (5) all options.

Results

Our results are listed in Table 4.8. We do not have sufficient evidence to conclude that the difference between any two configurations was not simply the result of chance.

Discussion

Since we were unable to discern any significant differences at all between the five combinations of options that we tested, we can conclude that we either need bigger samples in order to see the difference or simply that there is no difference. Since all times were fairly low, the experiment should perhaps be repeated with a more difficult instance.

Taking into account that preprocessing and simplification was designed to help solvers cope with "bad" (unoptimised) encodings, we could take this as an encouraging sign that our encoding does not contain any obvious redundancies.

An interesting object of further study is whether preprocessing and simplification make a difference for the other encodings that we considered in section 4.5. If preprocessing/simplification is effective for the other encodings, it could help explain *why* they are slower than our hand-crafted encoding.

4.9 Branching heuristics

Branching heuristics. The topic of branching heuristics runs all the way back to the original papers for the DP and DPLL algorithms. The authors admit that the choice of which variable to branch on first could have a big influence of the efficiency of the algorithm: “It is possible that the choice of p to be eliminated first is quite crucial in determining the length of computation required to reach a conclusion: a program to choose p is used, but no tests were made to vary this segment of the program beyond a random selection, namely the first entry in the formula table” [Davis et al., 1962].

Since then, several different heuristics were used [Marques-Silva, 1999]; the BOHM, MOMS, and Jaroslaw-Wang heuristics all attempt to (1) pick variables that occur frequently as a way to reduce the size of the instance as much as possible and speed up further computation; and (2) satisfy short clauses first, since (intuitively) short clauses are harder to satisfy than long clauses. Assuming that each variable is equally likely to be true or false, a longer clause has a greater probability of being satisfied than a short clause.

When the simple backtracking DPLL search evolved (at the turn of the millennium) into the modern algorithm of conflict-driven clause learning (CDCL) with its fast unit propagation [Marques-Silva and Sakallah, 1999; Moskewicz et al., 2001], these heuristics became too expensive to compute for every decision made by the solver. In [Moskewicz et al., 2001], the authors introduce the extremely successful and one of the first dynamic heuristics, Variable State Independent Decaying Sum (VSIDS).

The VSIDS heuristic is set apart from earlier heuristics primarily in that it does not have to be recomputed for every decision level; instead, the solver can simply pick the literal or variable with the currently highest score. Scores are also not reset when the solver backtracks, but updated continuously as the solver encounters new conflicts. The heuristic is widely cited as being *dynamic*, since it attempts to pick variables which were encountered during conflict analysis of recent conflicts first. Newer variants (for example the ones implemented in **MiniSat** and **clasp**) differ from the original VSIDS heuristic in keeping scores per variable rather than per literal and updating scores in a more efficient way.

One important parameter of the VSIDS heuristic is the variable activity *decay factor*, δ. During conflict analysis, the score of each variable is “bumped”, i.e. it is increased by a certain amount, the activity *increment*. The activity increment is given as δ^{-n} where n is the conflict number; the activity increment is an increasing sequence and this is the mechanism that ensures that recent conflicts dominate earlier conflicts. δ is a number between 0 and 1 (exclusive), and we see that values closer to 0 will give higher priority to newer conflicts while values closer to 1 will give old and new conflicts equal priority.[3]

For **MiniSat**, the default decay factor is $\delta = 0.95$. In the first experiment, we will measure the mean running time for values from 0.80 to 0.99 to see whether an optimal value exists. If the optimal value is closer to 0.80, it would mean that the instance is solved faster when more priority is given to newer conflicts (i.e. the heuristic is more dynamic), while if the optimal value is closer to 0.99, it would mean that the instance

[3]In practice, the values of the variable scores and the activity increment will overflow the hardware registers. Implementations typically overcome this problem by scaling down all the scores whenever an overflow would have occured.

is solved faster when priority is given more equally to past and current conflicts (i.e. the heuristic is less dynamic).

In [Goldberg and Novikov, 2002], the authors introduce a heuristic similar to VSIDS. In addition to keeping track of scores reflecting participation in recent conflicts, the heuristic selects the variable with the highest score among the most recently learned unsatisfied clause first. The heuristic was implemented in the **Berkmin** solver and is typically referred to as the Berkmin heuristic.

In [Ryan, 2004], the author introduces another heuristic called Variable Move-To Front (VMTF). This heuristic also tries, like the Berkmin heuristic, to be even more dynamic than VSIDS by branching on literals in recently learned conflict clauses. The heuristic maintains a list of literals where the first literal is always the next to be branched on. Initially, the list is sorted such that literals which occur frequently are closer to the front of the list. When a new conflict clause is derived, some fixed, small number of literals from the clause is moved to the front.

In [Simons et al., 2002], the authors describe the Unit heuristic. This heuristic attempts to minimise the size of the remaining search space by minimising the number of unassigned variables. This is essentially a lookahead heuristic; it asks, for each literal l, how many variables could we get rid of by assuming l? We see that this heuristic is closer in spirit to the BOHM, MOMS, and Jaroslaw-Wang heuristics, and likely very expensive to compute compared to the VSIDS, Berkmin, and VMTF heuristics.

The solver **clasp** implements all of these heuristics: the original (random) heuristic of the DP/DPLL procedures, VSIDS, Berkmin, Unit, and VMTF. This gives us the chance to compare the heuristics in a neutral setting (barring any peculiarities of the implementations) in the context of SHA-1 preimage attacks.

Polarity caching. The first implementation of the VSIDS heuristic kept track of scores for each literal rather than for each variable. This is not the case for newer solvers. Instead, they first pick a variable to branch on, then they decide which branch to try first. It has been observed that for many problems, most solutions are sparse in the sense that only a few variables of the model are 1. Therefore, most solvers employ the strategy of branching on 0 first.

In [Pipatsrisawat and Darwiche, 2007b], the authors noted that in instances with disjoint components (i.e. sets of clauses which do not share any variables), SAT solvers display suboptimal behaviour: the solver could spend a lot of time finding the solution to one affine subproblem, only to make a bad decision involving a variable from a different subproblem. Because of backjumping, the solver will skip over (and thus in a sense lose) the solution it found to the first subproblem. Keep in mind that subproblems do not necessarily need to exist in the original formula for this to happen; by branching on some variables near the root of the search, the solver could partition the instance into sets of (unsatisfied) clauses that do not share any (undefined) variable.

Their solution to the problem was to introduce the concept of *polarity caching*. The idea is to save the value of a variable when the variable becomes undefined. If the solver later branches on that variable, it will use the saved value. Thus it may solve the same subproblem again, but this time very quickly.

For SHA-1, we expect that polarity caching will not make a very big difference in the time it takes to solve an instance. This is because hash functions are designed to

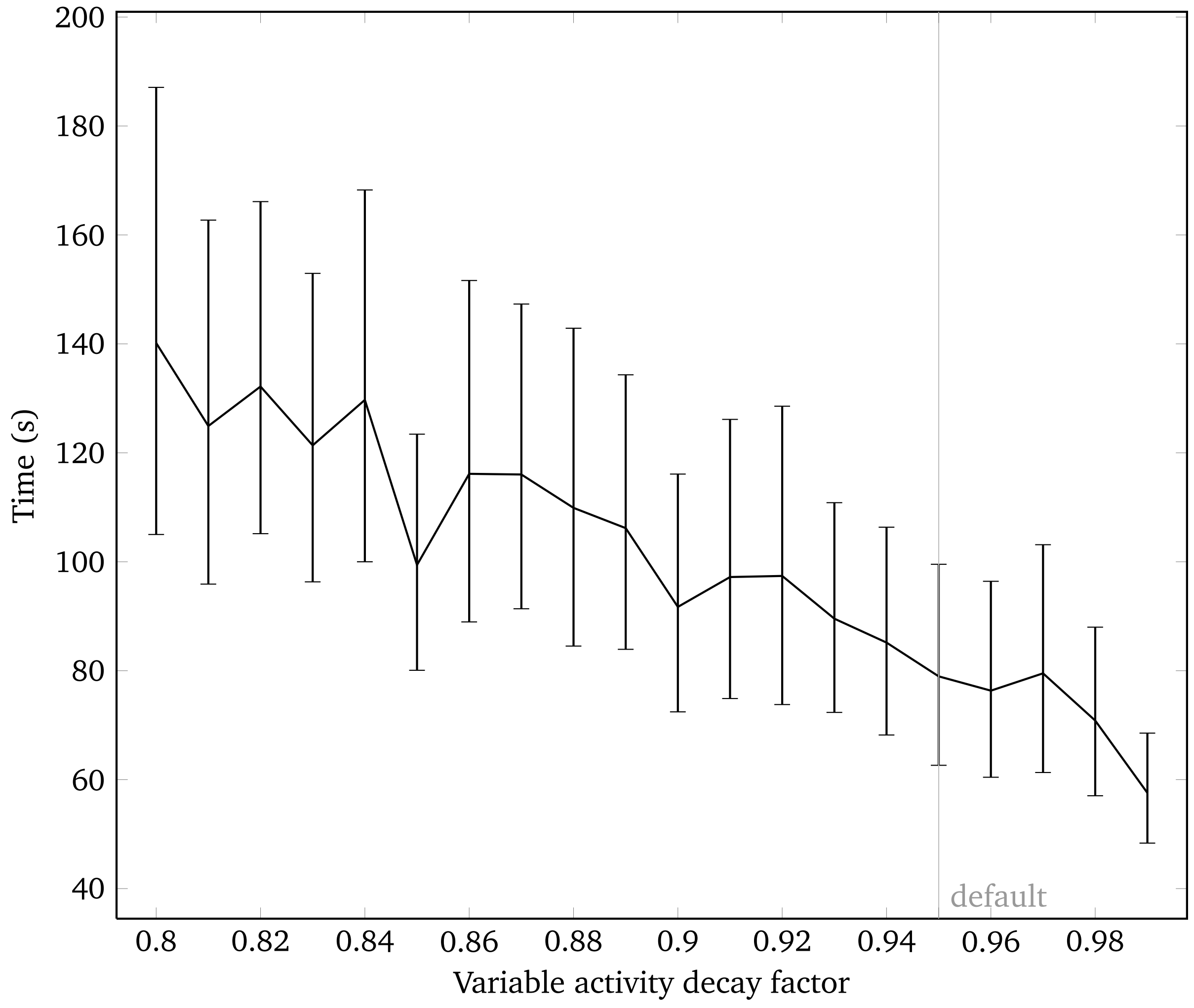


Figure 4.10: The mean (with uncorrected 95% confidence interval) time to find a 21-round preimage with minisat as a function of the variable activity decay factor.

exhibit the avalanche effect; flipping a single bit of the input should flip each bit of the output independently with probability 1/2. Therefore, when the solver backjumps and tries the alternative branch of the wrong decision, it is likely that any cached polarities will no longer be relevant.

In the third experiment, we will run **MiniSat** with its three modes of polarity caching: none, limited (only variables which were implied as a consequence of the very last decision will have their cached polarities updated), and full.

Results

In Figure 4.10 we show how the mean running time of the solver varies as a function of the variable activity decay factor. There is a clear trend towards larger values making the instance easier to solve; the best measurement was obtained for $\delta = 0.99$.

In Table 4.9 and Table 4.10 we compare all the branching heuristics implemented in **clasp**. We find that all differences are significant and conclude that the Berkmin

Table 4.9: The mean running times (with uncorrected 95% confidence intervals) for different branching heuristics in **clasp**. The heuristic **None** did not solve a single instance.

Heuristic	n	Mean (s)	Mean 95% CI (s)	
Berkmin	100	98.08	77.77	154.86
Vsids	100	291.93	224.82	400.97
Vmtf	100	922.24	673.56	1,277.39
Unit	50	12,615.14	8,853.95	21,949.02
None	—	—	—	—

Table 4.10: All pairwise significance tests on the the differences between the mean running times (the Games-Howell procedure) for different branching heuristics in **clasp**. Each sample has the size $n = 100$, except **Unit**, where $n = 50$ due to time constraints.

Heuristic 1	Mean (s)	Heuristic 2	Difference (s)	p-value
Berkmin	98.08	**Vsids**	193.86	**0.000**
		Vmtf	824.16	**0.000**
		Unit	12,517.06	**0.000**
Vsids	291.93	**Vmtf**	630.30	**0.000**
		Unit	12,323.20	**0.000**
Vmtf	922.24	**Unit**	11,692.90	**0.001**

heuristic is clearly superior to the other heuristics (as they are implemented in **clasp**), solving the instance in less than half the time of the next best heuristic, VSIDS. For the Unit heuristic, we had to lower the number of samples to $n = 50$ because it was taking too long. For the None heuristic, the solver did not finish even a single run and we aborted the attempt after 24 hours.

According to Table 4.11, we do not have enough evidence to say that any polarity caching mode is different from any of the others.

Discussion

We obtained a best value for the variable activity decay factor at $\delta = 0.99$. This indicates that the instance is solved faster when priority is given more equally to past and current conflicts (i.e. there is "less dynamicity"). However, we also found that the Berkmin heuristic, which is supposedly *more* dynamic, was better than the VSIDS heuristic.

Considering the difference between the Berkmin and VSIDS heuristics, it would be very interesting to see if an implementation of the Berkmin heuristic in **Minisat** could give similar improvements to the running time. We find one explanation in the

Table 4.11: Mean running times (with uncorrected 95% confidence intervals) and all pairwise significance tests on their differences (the Games-Howell procedure) for the three possible phase-saving settings. Each sample has the size $n = 100$.

Option 1	Mean (s)	Mean 95% CI (s)		Option 2	Difference (s)	p-value
full	73.07	61.10	90.36	limited	2.79	0.230
				none	20.54	0.139
limited	75.86	63.58	92.11	none	17.75	0.960
none	93.62	79.14	111.12			

literature of why the Berkmin heuristic performs so much better: "... another important advantage of Berkmin's heuristic over VSIDS: newly assigned variables tend to embrace ... variables whose joint assignment increases the chances of both quickly reaching a conflict in an unsatisfiable branch and satisfying out 'problematic' clauses in satisfiable branches" [Dershowitz et al., 2005]. The same paper proposes yet another heuristic, called the Clause-Based Heuristic (CBH), which they claim is better than the Berkmin heuristic. It would be interesting to see if CBH could offer even bigger improvements for our instances.

The fact that we could not obtain statistically significant differences when using different polarity caching settings could indicate that SHA-1 indeed does not lend itself to this type of optimisation. On the other hand, the difference between full polarity caching and no polarity caching seems large. To settle the question of whether polarity caching in fact *does* improve the running time, the experiment should be repeated with a more difficult instance and a larger sample size.

4.10 Restart heuristics

Simple combinatorial search algorithms (such as DPLL) sometimes exhibit a phenomenon known as heavy-tailed behaviour. This means that the algorithm's running time distribution exhibits a so-called heavy tail, or, in other words, that there is a small probability that the solver will run for a very long time before finding a solution. It turns out that it is possible to avoid these very long running times by occasionally restarting the search from the beginning and making a different set of choices (i.e. branching on different variables in a SAT solver). This works because it "prevents the procedure from getting trapped in the long tails" [Gomes et al., 1998].

The strategy used in [Gomes et al., 1998] was to restart the search after a fixed number of conflicts. This was also the strategy adopted by the early CDCL solvers. A geometric restart strategy was suggest in [Walsh, 1999] and adopted in an early version of **MiniSat**. In [Huang, 2007], the author demonstrated the utility of a restart strategy based on the Luby sequence [Luby et al., 1993]. This strategy was subsequently adopted by most modern CDCL solvers, including **MiniSat**.

The Luby restart strategy implemented in **MiniSat** is characterised by two parameters: the first restart interval k and the restart interval factor b. The restart

Table 4.12: Running times for **MiniSat** using the Luby and geometric restart strategies. Each sample has the size $n = 100$. The difference between the two configurations is significant (with a p-value of $p = 0.027$).

Configuration	Mean (s)	Mean 95% CI (s)	
Luby	68.41	58.72	80.85
geometric	112.98	83.61	162.26

interval is the number of conflicts that the solver will analyse before performing a full restart and is given as $k \cdot b^{\log_2 L_i}$, where $L = (1, 1, 2, 1, 1, 2, 4, \dots)$ is the Luby sequence and i is the restart number. By default, **MiniSat** uses the parameters $k = 100$ and $b = 2$.

In the literature, we find several recommendations for the use of small restart intervals: "... frequent restarts in combination with saving and reusing the previous phase can speed up SAT solvers on industrial instances tremendously, particularly on satisfiable ones" [Biere, 2008]; "Note that our proof requires the solver to restart at every conflict. While no actual solver utilizes this particular restart policy, the proof suggests that a frequent restart policy might be a key to the efficiency of modern solvers" [Pipatsrisawat and Darwiche, 2009]; and "The optimal restart strategy for this test set seems around a unit run of 6 or 8" [Haim and Heule, 2010].

In this experiment, we will first compare the geometric and Luby restart strategies, which are both implemented in **MiniSat**. Then we will try different values for the first restart interval and restart interval factor parameters of the Luby restart strategy in **MiniSat** in order to find the optimal values for our instance. Since we don't really know what to expect for the solving times for the first restart interval and the restart interval factor, these experiments are somewhat exploratory in nature. We do not define the range of the parameters that we test in advance, but will adjust them based on the observed results. Consequently, we cannot make hypotheses for specific tests (such as a comparison between the mean running time for the default value of a parameter and the mean running time for the observed best value of a parameter) in advance. Instead, the results of these experiments can be used to form specific hypotheses, but *new* samples (for the specific values that we want to compare) should be obtained before the hypothesis test is carried out.

Results

As listed in Table 4.12, we obtained the mean running times 68.41 s for the Luby restart strategy and 112.98 s for the geometric restart strategy. According to the Games-Howell test, the difference between the means is significant with a p-value of $p = 0.027$. The cactus plot in Figure 4.11 shows the difference between the distributions.

In Figure 4.12 we show the mean running time as a function of the restart interval factor k. The two measurements closest to the default value of $k = 100$ were 90.72 s at $k = 64$ and 84.86 s at $k = 128$. The smallest observed mean was 46.39 s at $k = 16384$. The (uncorrected) confidence intervals do not overlap.

In Figure 4.13 we show the mean running time as a function of the first restart interval b. For the default value of $b = 2$ we obtained a mean of 77.55 s, while the smallest observed mean was 60.47 s at $b = 4$.

Discussion

We have showed that the Luby restart strategy indeed performs better than the geometric strategy. This confirms that the generally held belief also holds for our specific instance.

In the case of restart intervals, however, our results contradict the assertions of [Biere, 2008; Pipatsrisawat and Darwiche, 2009; Haim and Heule, 2010] that shorter restart intervals lead to better solving times; [Haim and Heule, 2010] claims that the best times were obtained for $k = 6$ and $k = 8$, while in our experiments the best times were obtained for $k = 16,384$. For the restart interval factor, the results are not as clear. The best time we obtained was for $b = 4$, which indeed also increases the restart intervals; however, it seems the effect of varying the value of b is less dramatic than that of varying the value of k. In any case, we can safely conclude that *larger* restart intervals, rather than shorter, are better for our instances.

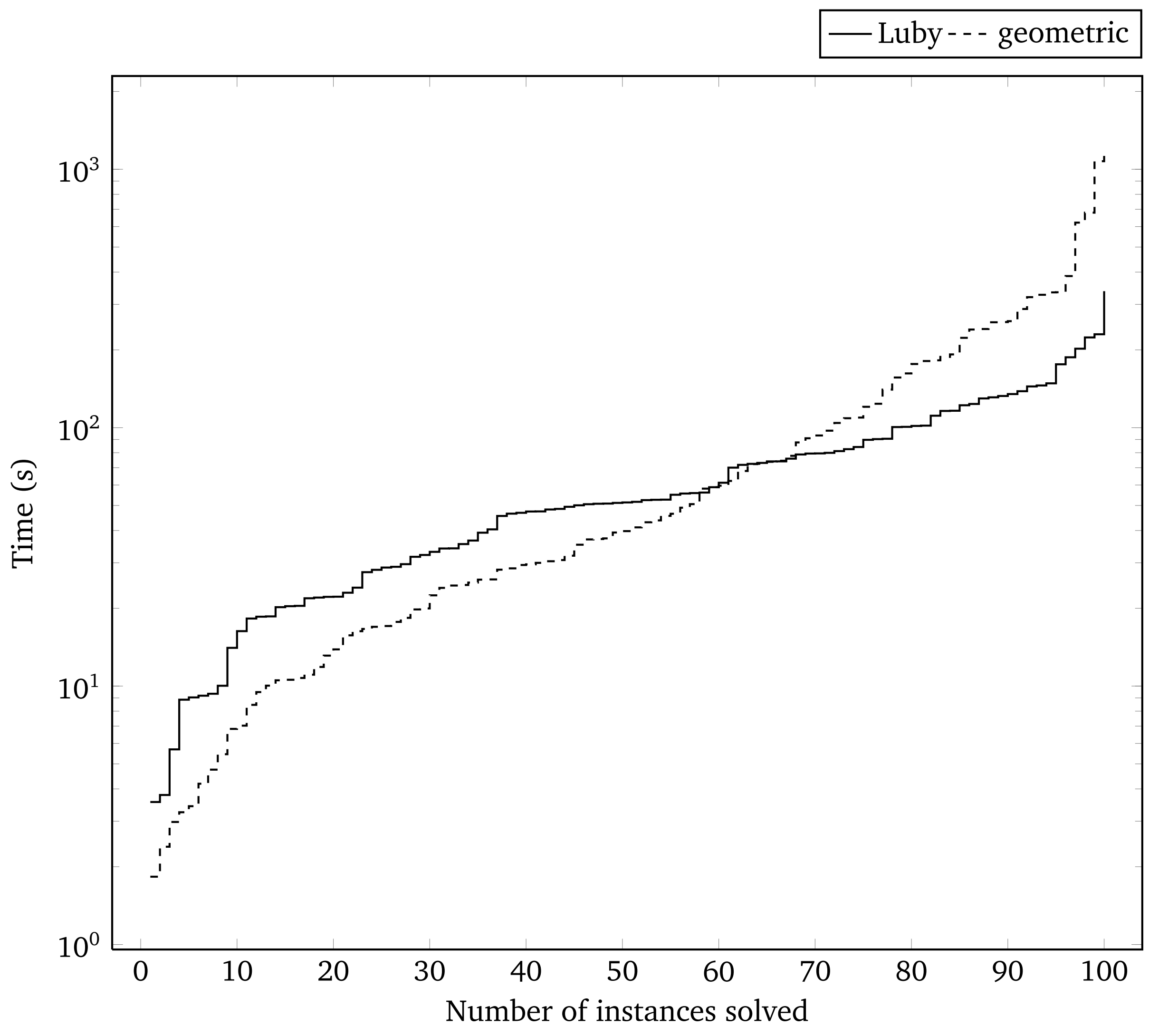


Figure 4.11: “Cactus plot” showing the time it takes to solve 100 instances for Luby vs. geometric restart strategies using **MiniSat**. The distribution of running times for the Luby restart strategy clearly has a sharper peak while maintaining the overall shape; note, however, that the logarithmic scale means that much more time is saved in the right tail than is lost in the left tail.

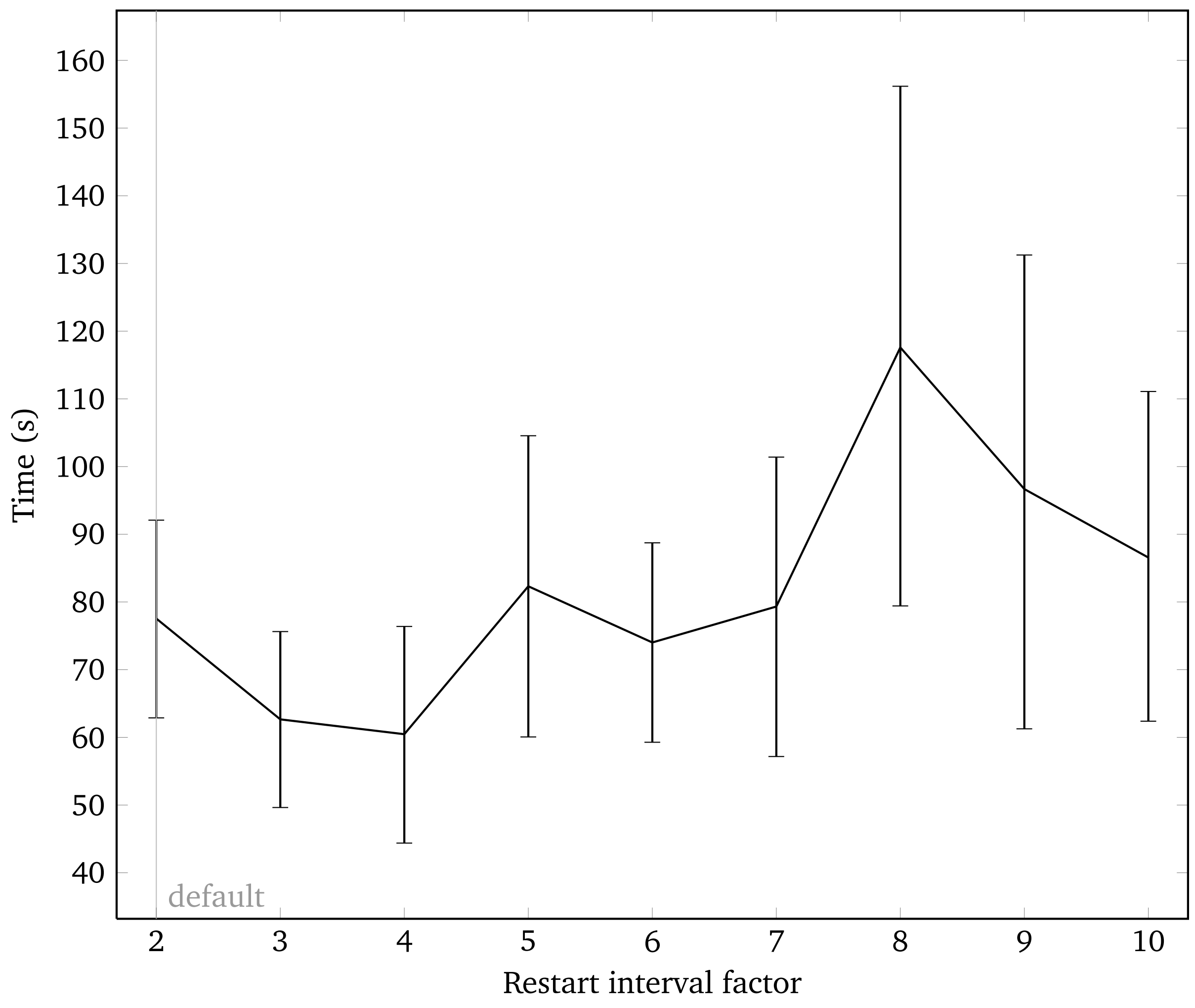


Figure 4.12: The mean time (with uncorrected 95% confidence intervals) to find a 21-round preimage with minisat as a function of the restart interval factor b.

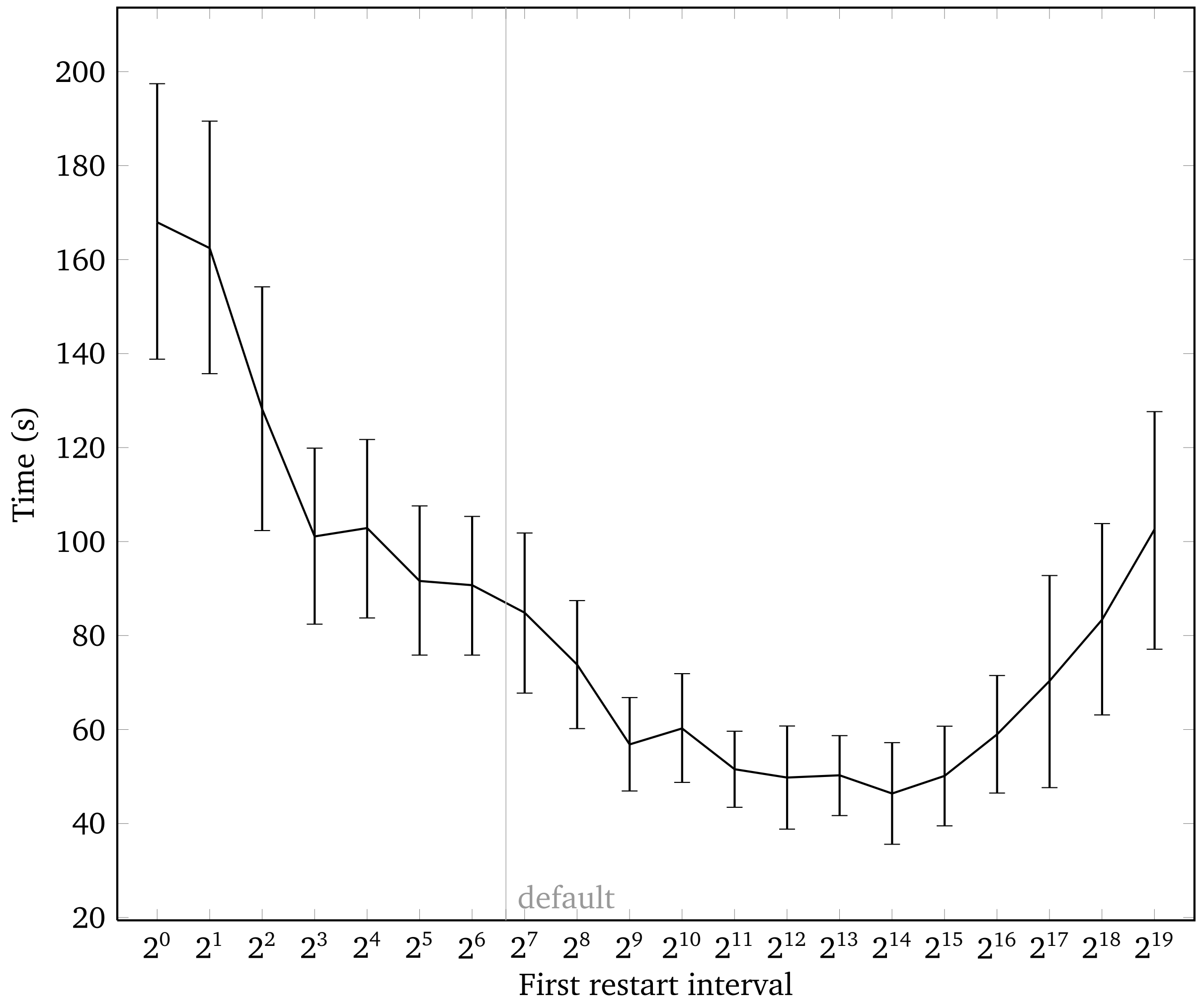


Figure 4.13: The mean time (with uncorrected 95% confidence intervals) to find a 21-round preimage with minisat as a function of the first restart interval k.

4.11 Conflict analysis

We have already explained the mechanism of conflict analysis. Several extensions and improvements have been suggested since its introduction in [Marques-Silva and Sakallah, 1999]. In this experiment, we aim to explore the potential of some of these additional techniques.

Conflict clause minimisation was suggested in [Sörensson and Eén, 2002] after an observation that many conflict clauses can be made smaller by "resolving out" certain literals (because the literal in question was propagated from a clause which, when resolved with the conflict clause, subsumes the conflict clause). A slightly more expensive version also handles the cases where the reason clause contains additional literals, but where the additional literals can be ignored because they would be implied. A more detailed description of these algorithm can be found in [Sörensson and Biere, 2009]. These two clause minimisation techniques are implemented in **MiniSat** and can be enabled using the `--ccmin` option. There are three possible settings: 0 (no minimisation), 1 (cheap minimisation), and 2 (expensive minimisation).

During unit propagation, it is entirely possible that the same literal is implied by multiple clauses. It is even possible that a decision variable is implied by later decisions. However, most SAT solvers take into account only the first implication they encounter, and completely ignore any additional implications. An extended notion of implication graphs where all implications are taken into account was proposed in [Audemard et al., 2008]. Extra implications are called "inverse arcs" in their terminology. Although it is possible to use inverse arcs to generate multiple new learnt clauses, they only use the extra information to possibly shorten the conflict clause and derive a better backjumping level. This feature has been implemented in **clasp** and can be enabled using the `--reverse-arcs` option. There are four possible settings: 0 (don't resolve any reverse arcs with the conflict clause), 1 (allow at most one literal to be removed using inverse arcs), 2 (allow at most half the literals in the conflict clause to be removed using inverse arcs), 3 (allow any number of literals in the conflict clause to be removed using inverse arcs).

We will run the respective solvers with all the different settings on instances with 21 rounds and 160 fixed hash bits to find out if they improve the average running time or not.

Results

Our results are listed in Table 4.13 and Table 4.14. In short, we find no significant difference between any pair of settings.

Discussion

Since none of our tests were significant, we cannot reject the hypothesis that there is in fact no difference between the various settings of the two techniques. However, it is still possible that these techniques are more useful for more difficult instances.

Table 4.13: Mean running times (with uncorrected 95% confidence intervals) and all pairwise significance tests on their differences (the Games-Howell procedure) for the 3 clause minimisation settings in **MiniSat**. The configurations are ordered by the mean running time. Each sample has the size $n = 100$.

Setting 1	Mean (s)	Mean 95% CI (s)		Setting 2	Difference (s)	p-value
2	74.79	62.87	89.59	1	0.40	0.999
				0	8.02	0.744
1	75.19	63.16	91.06	0	7.61	0.765
0	82.80	69.46	103.64			

Table 4.14: Mean running times (with uncorrected 95% confidence intervals) and all pairwise significance tests on their differences (the Games-Howell procedure) for the 4 conflict analysis settings in **clasp**. The configurations are ordered by the mean running time. Each sample has the size $n = 100$.

Setting 1	Mean (s)	Mean 95% CI (s)		Setting 2	Difference (s)	p-value
0	48.35	39.76	59.86	3	7.12	0.933
				1	18.01	0.325
				2	23.26	0.127
3	55.48	40.83	88.26	1	10.89	0.871
				2	16.14	0.671
1	66.36	51.39	88.82	2	5.25	0.978
2	71.62	57.18	95.21			

4.12 Learnt clause cleaning heuristics

While conflict analysis clause learning allows the solver to prune large parts of the search space, it also slows the solver down during unit propagation. Therefore, most solvers also try to identify the most useful learnt clauses and delete the rest. Lately, the topic of which clauses to keep has become an increasingly important topic in research on parallel solvers which need to know which clauses to share between threads. One simple heuristic is based on the size of the clause; shorter clauses are clearly better from a purely objective point of view, since they exclude a much larger part of the search space than longer clauses (a clause of length 1 fixes one variable to a particular value and therefore halves the size of the total search space; similarly, a clause of length n divides the total search space by 2^n).

The heuristic used in **MiniSat** was inspired by that of variable activities. Every time a clause participates in conflict analysis, its activity is multiplied by $\delta^- i$, where δ is the clause activity *decay factor* and i is the current conflict number. The clause activity

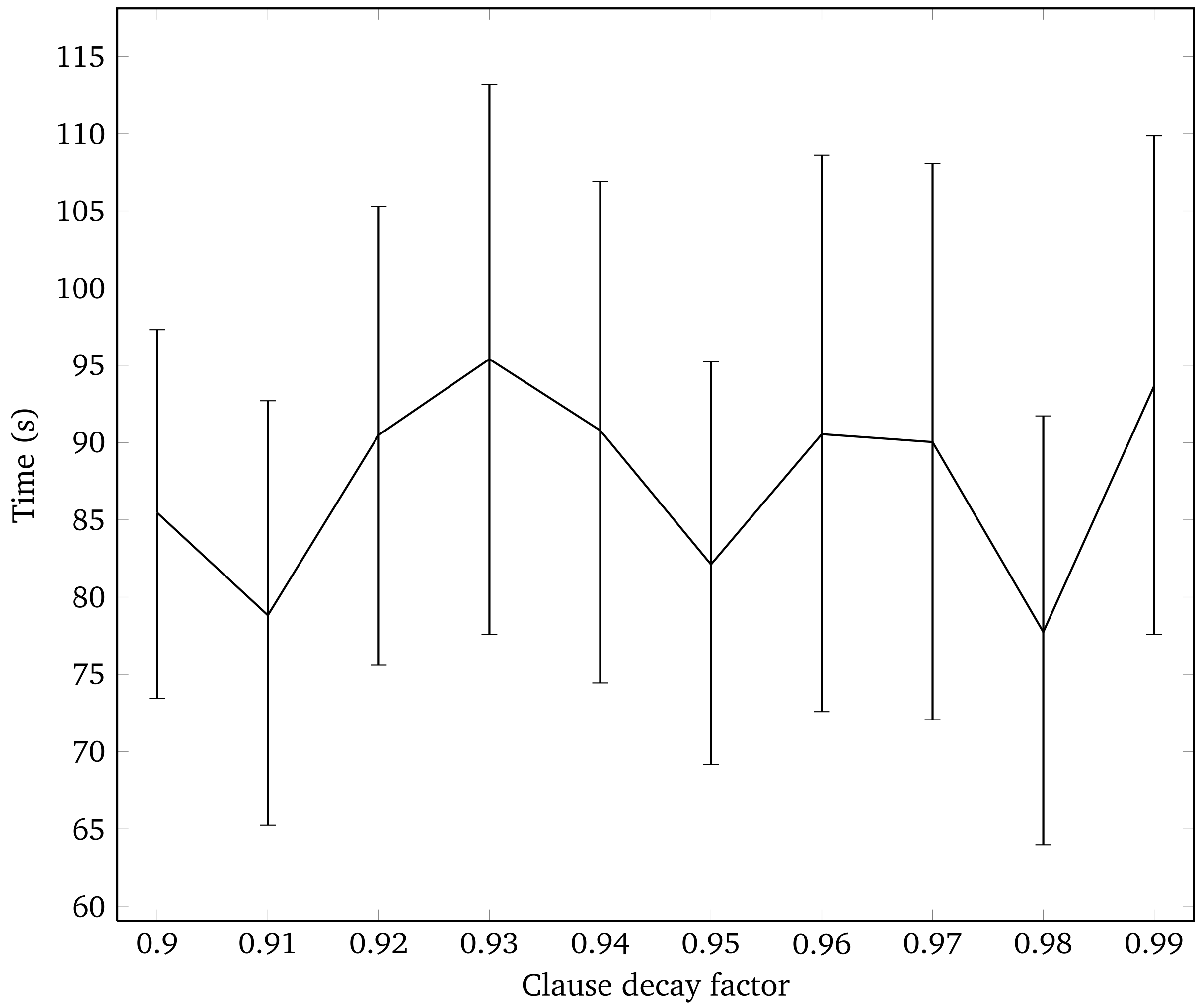


Figure 4.14: The mean (with uncorrected 95% confidence intervals) time to find a 21-round preimage with **MiniSat** as a function of the clause activity decay factor.

therefore intuitively corresponds to how much the clause has contributed to recent progress. Correspondingly, **MiniSat**'s clause cleaning heuristic periodically deletes clauses which have a low activity.

The object of this experiment is to determine if there exists an optimal value for δ. Possible values lie between 0 and 1. Just like for variable activities, a value close to 0 means that clauses that contributed to recent conflicts are prioritised over clauses that contributed to old conflicts, while a value closer to 1 means that the solver treats old and new conflicts more equally. By default, **MiniSat** uses a value of $\delta = 0.999$.

Results

Our results are shown in Figure 4.14.

Discussion

There does not appear to be any systematic change in the running time of the solver as a function of the decay factor and any variation observed is likely due to chance alone. This could indicate that our instances are not very sensitive to exactly which clauses are deleted.

4.13 Multiple solutions (nth-preimage attacks)

Some solvers have the possibility to search for more than one model. This is typically implemented by augmenting the instance with the negation of the current set of decisions that lead to the solution. For example, if the solution is $x_1 \land x_2 \land \neg x_3 \land x_4$, we can add the clause $\neg x_1 \lor \neg x_2 \lor x_3 \lor \neg x_4$ to the instance and continue the solving process. The newly added clause will prevent the same solution from being found again. This process can be repeated until we learn the empty clause (i.e. until the instance is no longer satisfiable).

Finding multiple solutions is essentially an nth-preimage attack, where the first solution is a preimage attack (with no restrictions on what the message may be), the second solution is a second-preimage attack (the message must be different from the first), etc.

The purpose of the following experiment is to find out whether it is easier or harder to find subsequent solutions. We might expect that it gets easier because the solver has (1) learnt clauses which are useful regardless of the specific hash value we are trying to find a preimage for; (2) established useful variable activities (and other statistics used in various heuristics); (3) entered a part of the search space where there are multiple solutions with a low Hamming distance (this is unlikely, however, assuming the pseudo-random property of SHA-1). If it gets easier to find subsequent solutions, we can try to use this to our advantage by finding out *why* it is faster to find subsequent solutions and supply this information to the solver before we search for the first solution. However, we might also expect that it gets harder, since the instance is getting more constrained as we exclude possible solutions.

The solver **clasp** has an option to search for any number of solutions before stopping the search. We will measure the time it takes to find 1, 2, 4, 8, and 16 solutions, and divide the mean time by the number of solutions found to obtain the mean time per solution. Because of time constraints, we do not try to find more than 16 solutions. We will use instances with 21 rounds, 160 fixed hash bits, and 0 fixed message bits.

Because of the problem with outliers in the running time when using **clasp** (see section 4.6; the problem is exacerbated by the fact that we are searching for multiple solutions), we also modify **MiniSat** to look for 16 solutions and output the time it took to find each solution.

Results

The results for **clasp** and **MiniSat** are given in Figure 4.15 and Figure 4.16, respectively.

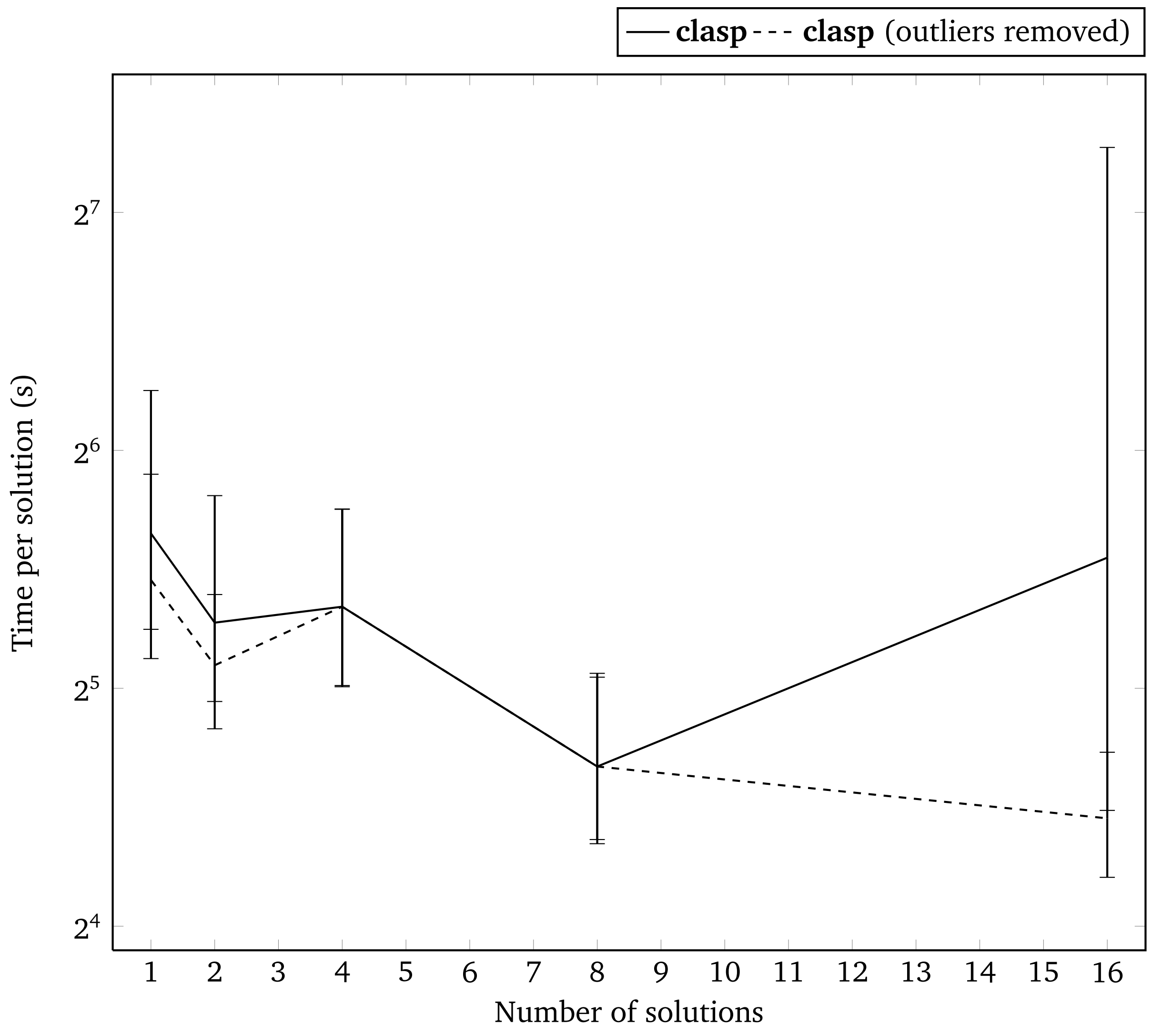


Figure 4.15: The mean (with uncorrected 95% confidence intervals) time to find multiple distinct 21-round preimages for the same hash using **clasp**. Each data point is independent of the others (i.e. each data point was calculated using 100 distinct runs).

Discussion

Looking at the results for **MiniSat**, there is no clear trend; the observed variation is likely due to chance. For **clasp**, the results are also somewhat ambiguous because of the outlier problem. If we assume that the outliers are due to an error in the implementation of the solver and discard these runs, there is a clear trend: subsequent solutions are easier to find.

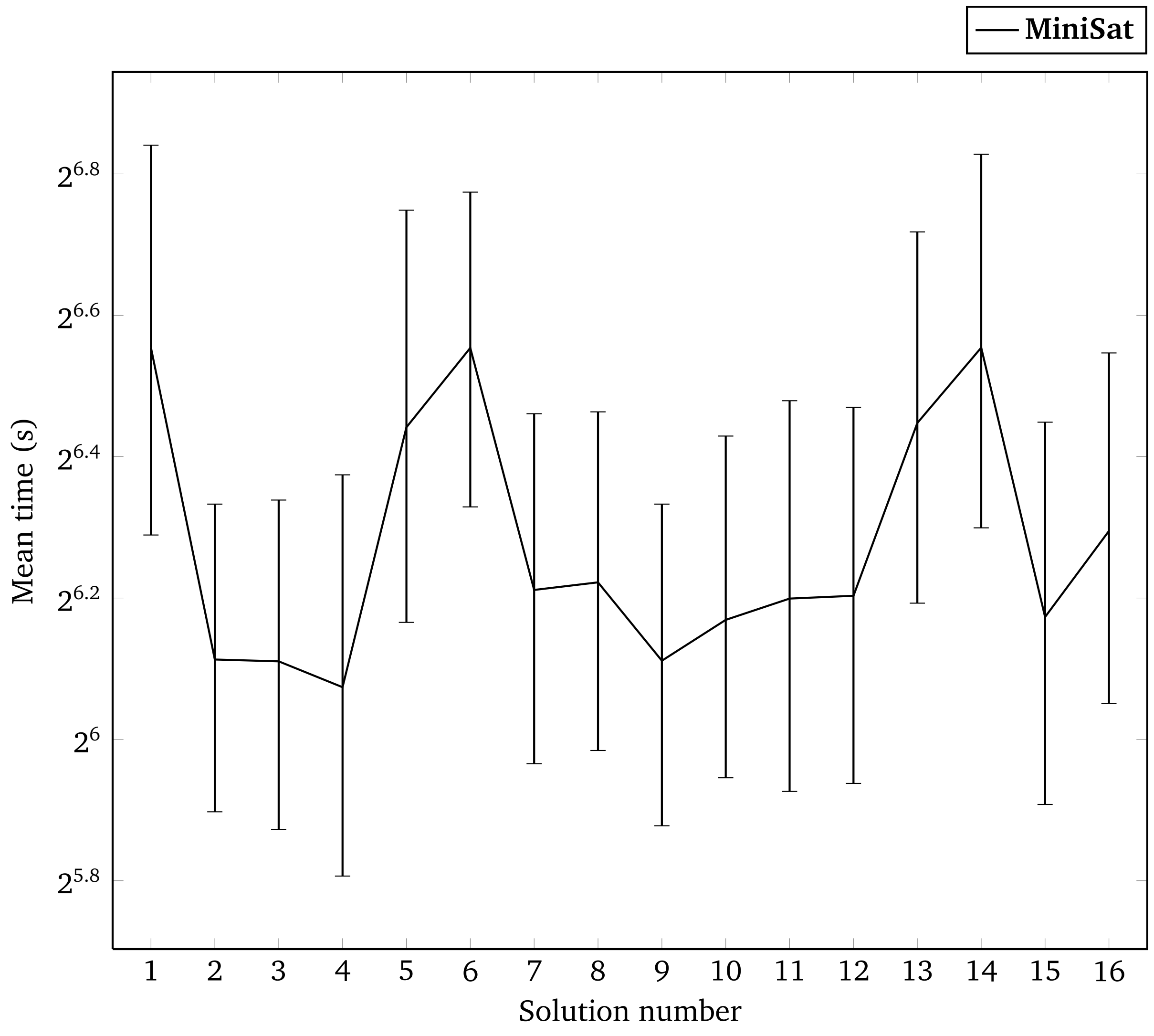


Figure 4.16: The mean (with 95% confidence interval) time to find multiple distinct 21-round preimages for the same hash using **MiniSat**. Each data point is *not* independent of the others (i.e. each data point was calculated using the same 100 runs).

4.14 Reusing learnt clauses

As we have already mentioned, clause learning prunes the remaining search space by guiding the search away from areas which are proven (using resolution) to contain no solutions. Clause learning is one of the cornerstones of the modern CDCL solver.

Clause learning is very closely connected with the concept of consistency in constraint set programming. Informally, arc consistency of a set of clauses means that there is no way to make a decision that will lead to a conflict (assuming unit propagation is carried out as usual). When a conflict *does* occur, the learnt clause will prevent the solver from making the same wrong decision again (since the learnt clause will propagate the last variable before the solver has a chance to branch on it). We can thus view clause learning as a process that brings the instance closer to a state of consistency.

Consistency is a desirable property, since it means that we can find solutions (and detect unsatisfiability) quickly. If we had a consistent encoding of SHA-1, we would essentially have a polynomial algorithm for finding preimages. However, such an encoding would probably yield a number of clauses exponential in the number of variables. In fact, we do have such an encoding already, we just have to enumerate all possible messages and their solutions. The whole instance would have $2^{512} \times 160$ clauses.

Regardless of whether we can achieve full consistency, clause learning still improves the propagativity of the instance. In our case, we can divide all learnt clauses into two categories: those which are specific to the particular hash value we are trying to find a preimage for, and those which are valid regardless of the hash value. In this experiment, we will try to identify those learnt clauses which are not specific to a particular hash value; in doing so, we can provide them directly to the solver as part of the input and (hopefully) allow the solver to focus on learning those clauses which are specific to the particular hash value.

The experiment consists of two parts: the clause learning part (where we try to identify universally valid clauses that are not explicit in the original encoding), and the experiment proper (where we measure the time it takes to solve an instance that includes the extra clauses).

In order to identify those clauses which are universally valid, we will assume the following proposition: any clause which is learnt by sufficiently many runs of the solver on instances with *different* hash values, chosen at random, is valid for *all* instances (i.e. it follows logically from the encoding of SHA-1 alone). In other words, if we run the solver on 100 instances encoding attacks on different hash values and e.g. 90 of those runs have learnt the exact same clause, we will assume that the clause is actually valid for all instances regardless of any specific hash value.

In order to obtain the extra learnt clauses, we use the solver **clasp** which comes with an option to output all learnt clauses before the solver exits. We set a time limit of 300 seconds and disable clause deletion so that we can better see *all* the clauses learnt by the solver (until a solution is found or the time limit is reached).

For the second part of the experiment, we will run **MiniSat** on instances which are augmented with (some of) the extra clauses learnt in the first step.

Results

Our results are depicted in Figure 4.17. Augmenting the instances with clauses which were learnt in 20 or more runs (out of the 100 runs), we found that the instance became unsatisfiable. Using the clauses which were learnt in 30 or more runs, we found that the running time increased significantly. For clauses which were learnt in 40 or more runs, 50 or more runs, etc., there was little or no difference in the average running time.

Discussion

The results are slightly disappointing and we have to wonder *why* supplying the extra clauses to the solver before starting the search did not improve the running time. One

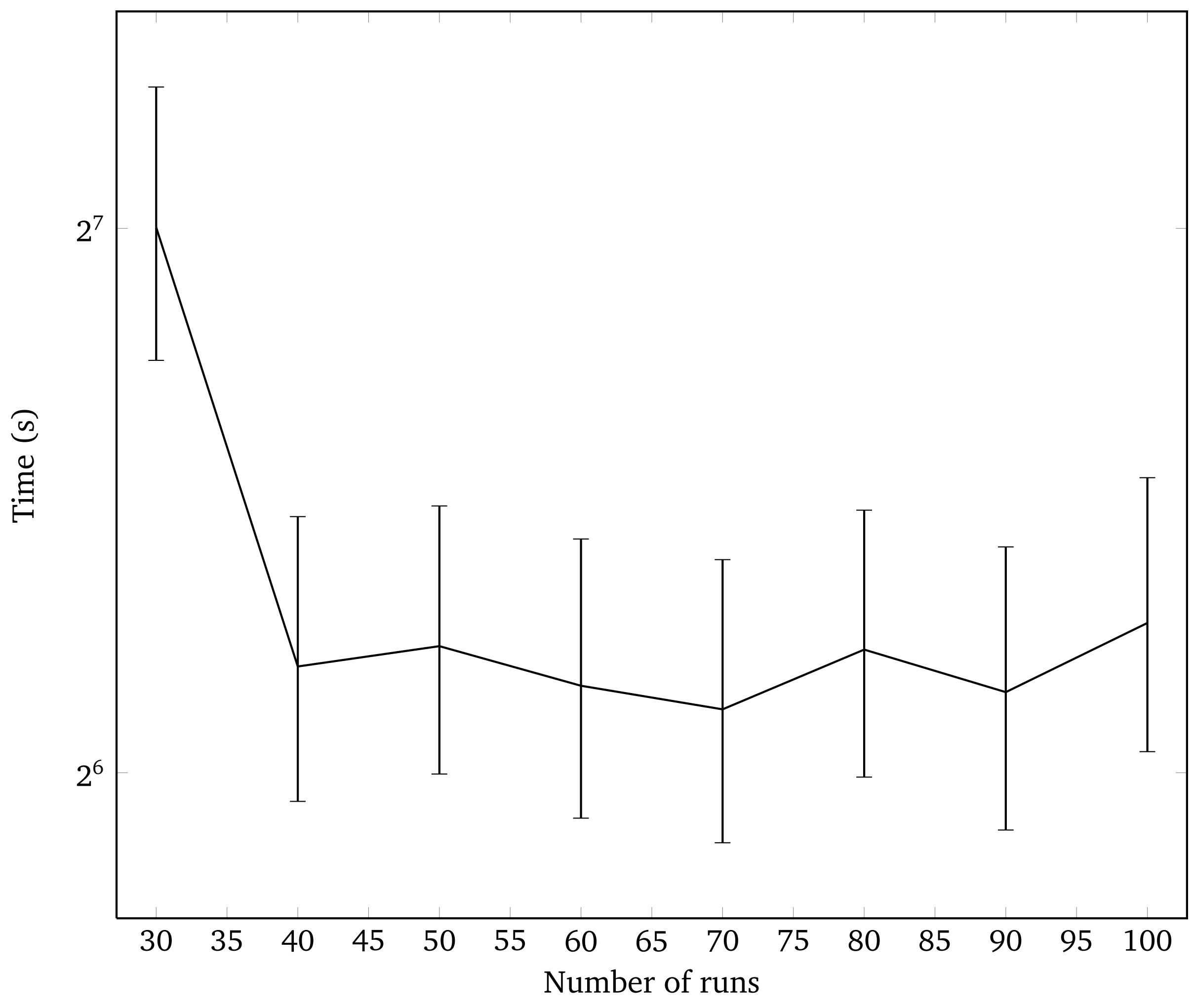


Figure 4.17: The mean running time (with uncorrected 95% confidence intervals) to solve an instance which has been augmented with clauses that were learnt in previous runs. We collected the learnt clauses from 100 runs. Each data point represents instances which include clauses which were found in more than a certain number of runs.

reason could be that the extra clauses induce an overhead during propagation.

We also refer to section A.2, which contains some statistics on the learnt clauses.

Chapter 5

Conclusions and future work

Our goal was to find ways to improve SAT-based attacks on SHA-1. Towards this goal, we have investigated many aspects of both the encoding and the solving processes. Our most important finding is that the encoding typically used in the literature (the straightforward use of the Tseitin transformation on a circuit representation) is suboptimal, and we have proposed a new "handcrafted" encoding which cuts the solving time by a factor of more than 2 compared to the best known encoding used in the literature (see section 4.5). In particular, our improved encoding of SHA-1 makes use of a new encoding of 32-bit modular addition based on pseudo-boolean constraints.

We have also studied the performance of the SAT solver as a function of the difficulty of the instance. One startling result is that fixing parts of the message (to values which are known to be part of at least one solution) actually makes the instance harder, even though the absolute size of the search space becomes smaller (see section 4.3). This indicates that the SAT solver is better able to find a solution when there are more possible solutions to find, regardless of the size of the search space. In light of these considerations, it becomes very interesting to consider attacks on multiple message blocks (i.e. two or more instances of the compression function). This could be a topic for further research.

In the area of SAT solving techniques, we have not made any substantial new findings. We examined variations of several popular heuristics for branching, restarts, and clause deletion. The most interesting result is probably the fact that longer restart intervals improve the mean running time of the solver (see section 4.10). This dirctly contradicts the claim found in recent literature that very short restart intervals are better. We also found it surprising that the BerkMin branching heuristic so outperformed the popular VSIDS heuristic as they are implemented in **clasp** (see section 4.9). It would be interesting to see whether we can get a similar improvement in **MiniSat** (which we found to be the fastest solver for the more difficult instances) by replacing its VSIDS heuristic with the BerkMin heuristic. Although it was shown that **MiniSat** beats **clasp** (with pseudo-boolean constraints), we believe that further research on the use of XOR and pseudo-boolean constraints, especially with regards to conflict analysis, could yield even better results.

Lastly, we note that our results on the encoding process and parameters of the solver likely transfer to any SAT problem that includes an encoding of SHA-1, e.g. collision attacks (which *are* a practical target for SAT-based techniques [McDonald et al., 2009]).

Appendix A

Variable and clause statistics

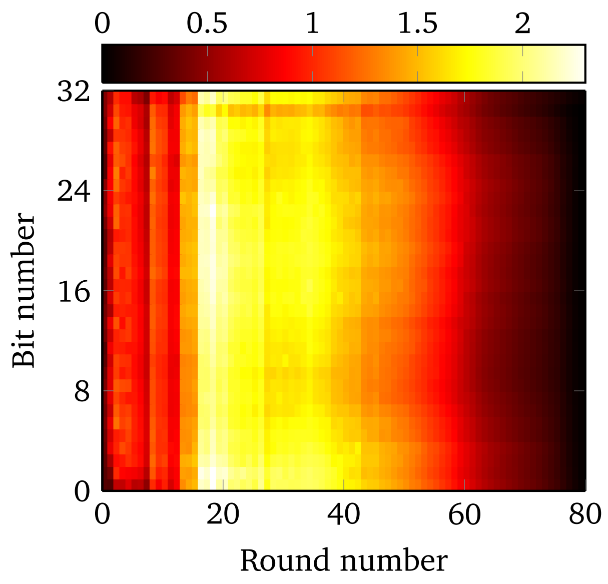
A.1 Per-variable statistics

In order to learn something about how the solver works internally, we modified **MiniSat** to count the number of times each variable is (1) used as a decision variable (“decisions”), (2) propagated (“propagations”), (3) the source of a conflict (“conflicts”), and (4) asserted while backtracking (“implications”). . We have four types of variables: (1) the message and message schedule words W; (2) the internal state a; (3) the carry bits c_0 and c_1 used in encoding addition; and (4) the per-round logical functions f.

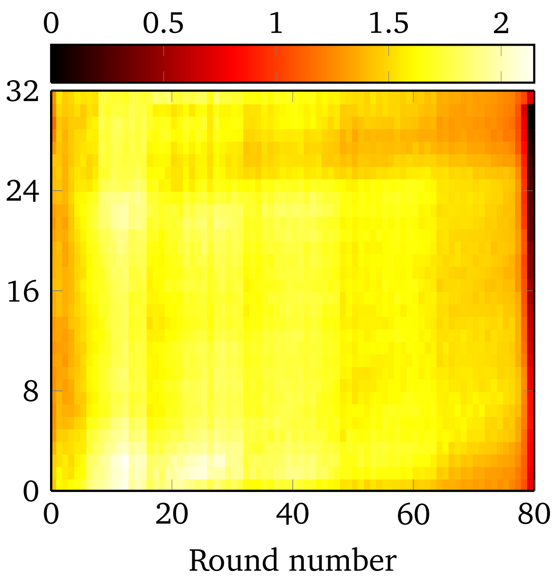
We ran the modified solver on 100 instances with 80 rounds and 4 fixed hash bits. For reference, the mean running time was 952.60 s, although some of this time was spent on updating the statistics we gathered.

In Table A.1, we show the distribution of each type of solver event over the different types of variables. We note that the percentages for conflicts and implications seem to be highly correlated. Also noteworthy is the fact that roughly 50% of conflicts and implications occur for variables encoding the message and the message schedule.

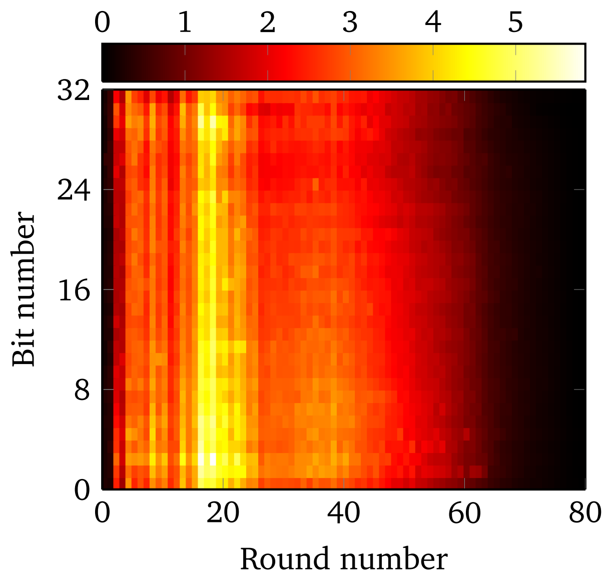
In Figure A.1, Figure A.2, Figure A.3, Figure A.4, and Figure A.5, we show more detailed statistics: for each bit in each word in each round, we indicate its activity in each type of solver event using heatmaps. We make a few observations: some of the heatmaps (e.g. decisions in Figure A.2) show a similar pattern in the bits across each word as we saw in section 4.4: 2–3 of the least significant bits and 5–6 of the most significant bits seem to behave differently from the middle bits. Also keep in mind that rounds 0–15 are special, since they operate directly on words of the message; rounds 0–19, 20–39, 40–59, and 60–79 use different per-round logical functions. These variations are observable: for example, very few decisions are made in the bits of the message (e.g. Figure A.1) compared to the immediately following rounds (from round 16 onwards).



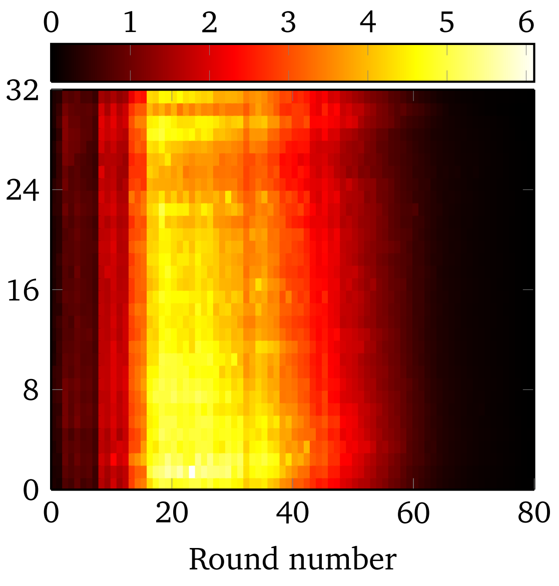
(a) Decisions



(b) Propagations

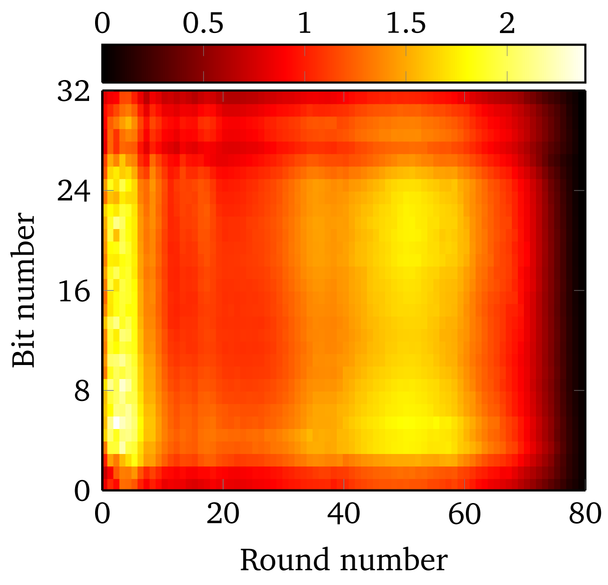


(c) Conflicts

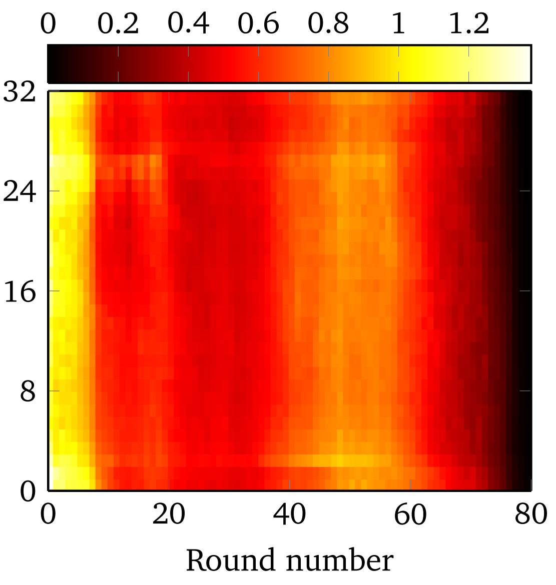


(d) Implications (when backtracking)

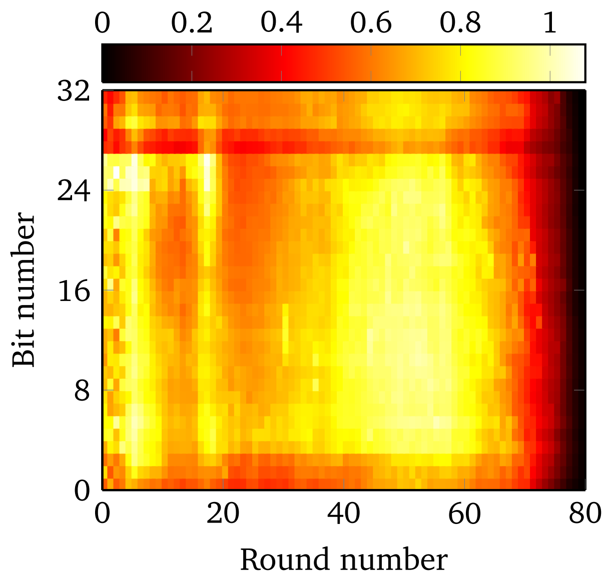
Figure A.1: Heatmaps showing how often (in parts per ten thousand) the variables encoding $W_{i,j}$ (bit j of the message schedule entry for round i) participate in the given activity when using **MiniSat** and the VSIDS branching heuristic.



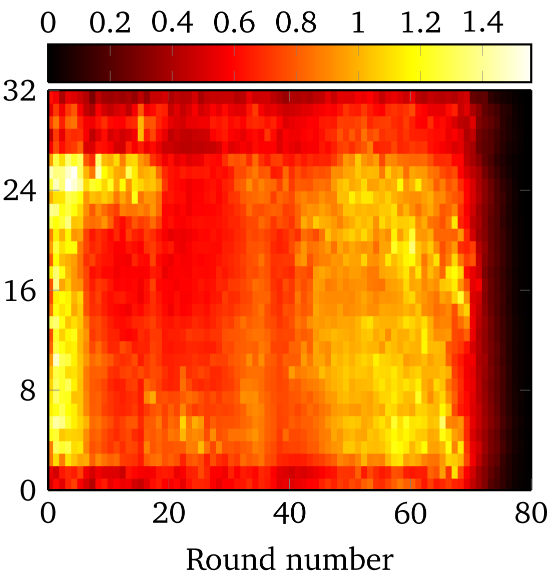
(a) Decisions



(b) Propagations

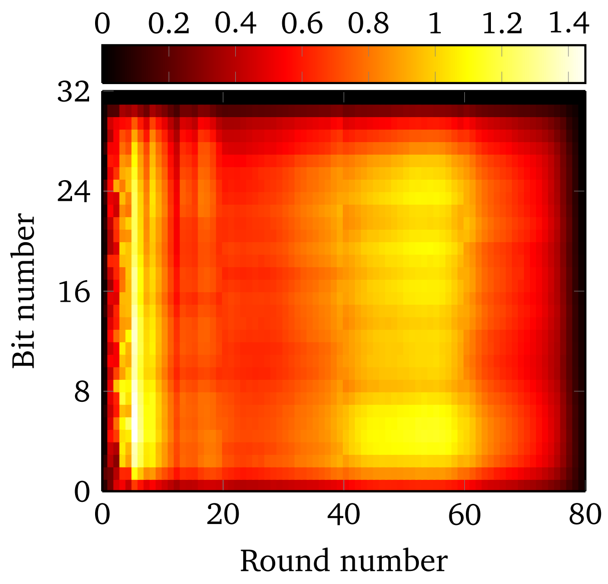


(c) Conflicts

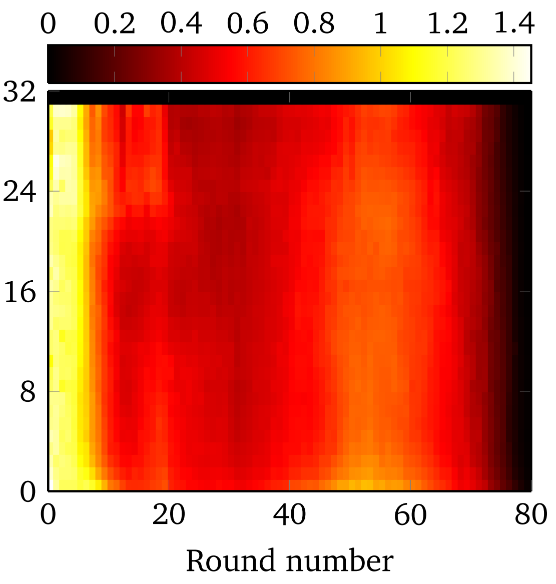


(d) Implications (when backtracking)

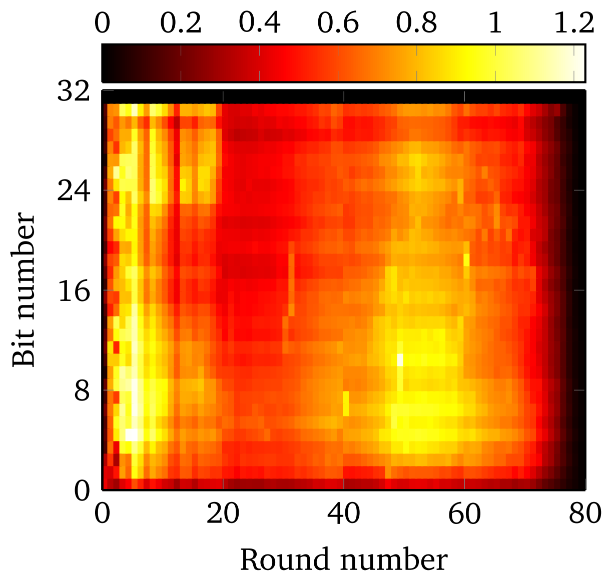
Figure A.2: Heatmaps showing how often (in parts per ten thousand) the variables encoding $a_{i,j}$ (bit j of state for round i) participate in the given activity when using **MiniSat** and the VSIDS branching heuristic.



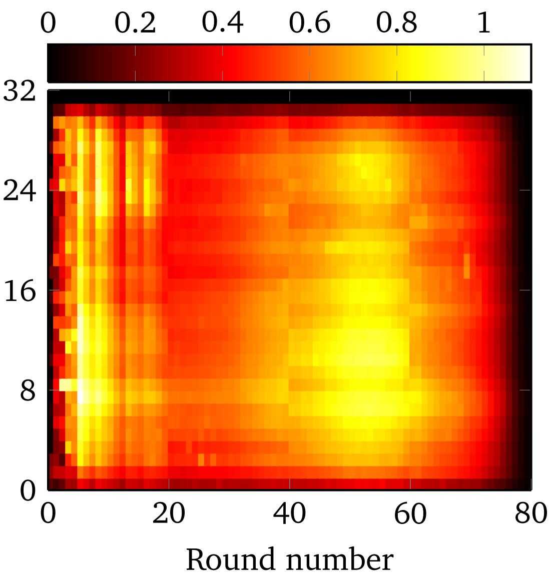
(a) Decisions



(b) Propagations

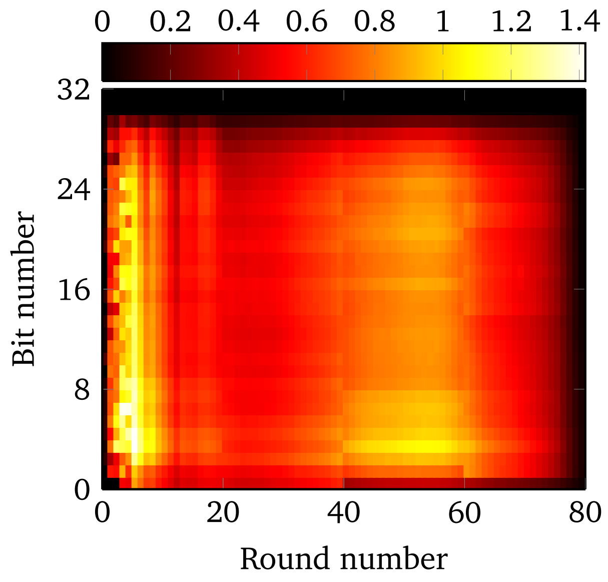


(c) Conflicts

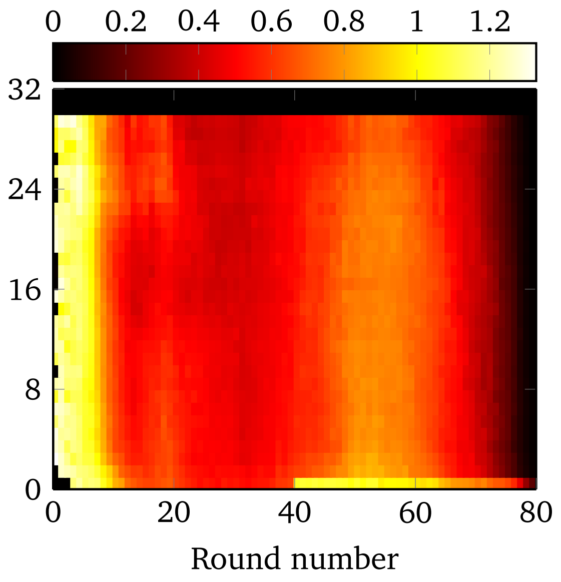


(d) Implications (when backtracking)

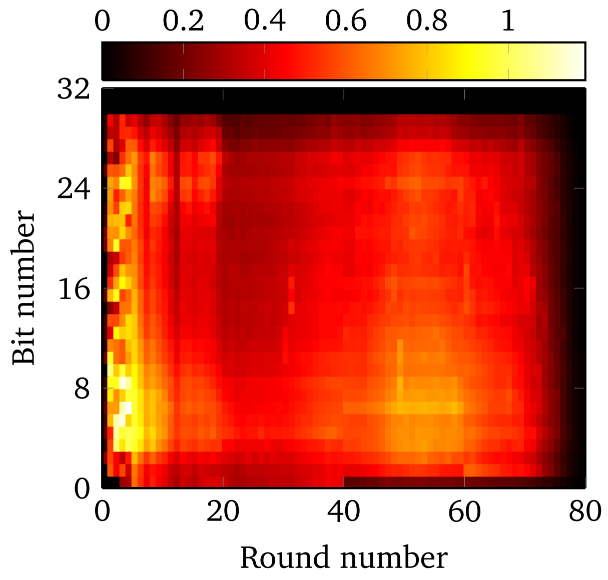
Figure A.3: Heatmaps showing how often (in parts per ten thousand) the variables encoding the first carry bit of the per-round adder participate in the given activity when using **MiniSat** and the VSIDS branching heuristic.



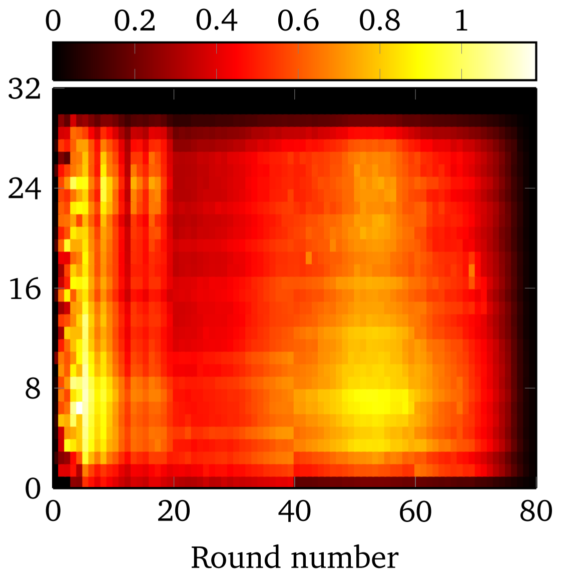
(a) Decisions



(b) Propagations



(c) Conflicts



(d) Implications (when backtracking)

Figure A.4: Heatmaps showing how often (in parts per ten thousand) the variables encoding the second carry bit of the per-round adder participate in the given activity when using **MiniSat** and the VSIDS branching heuristic.

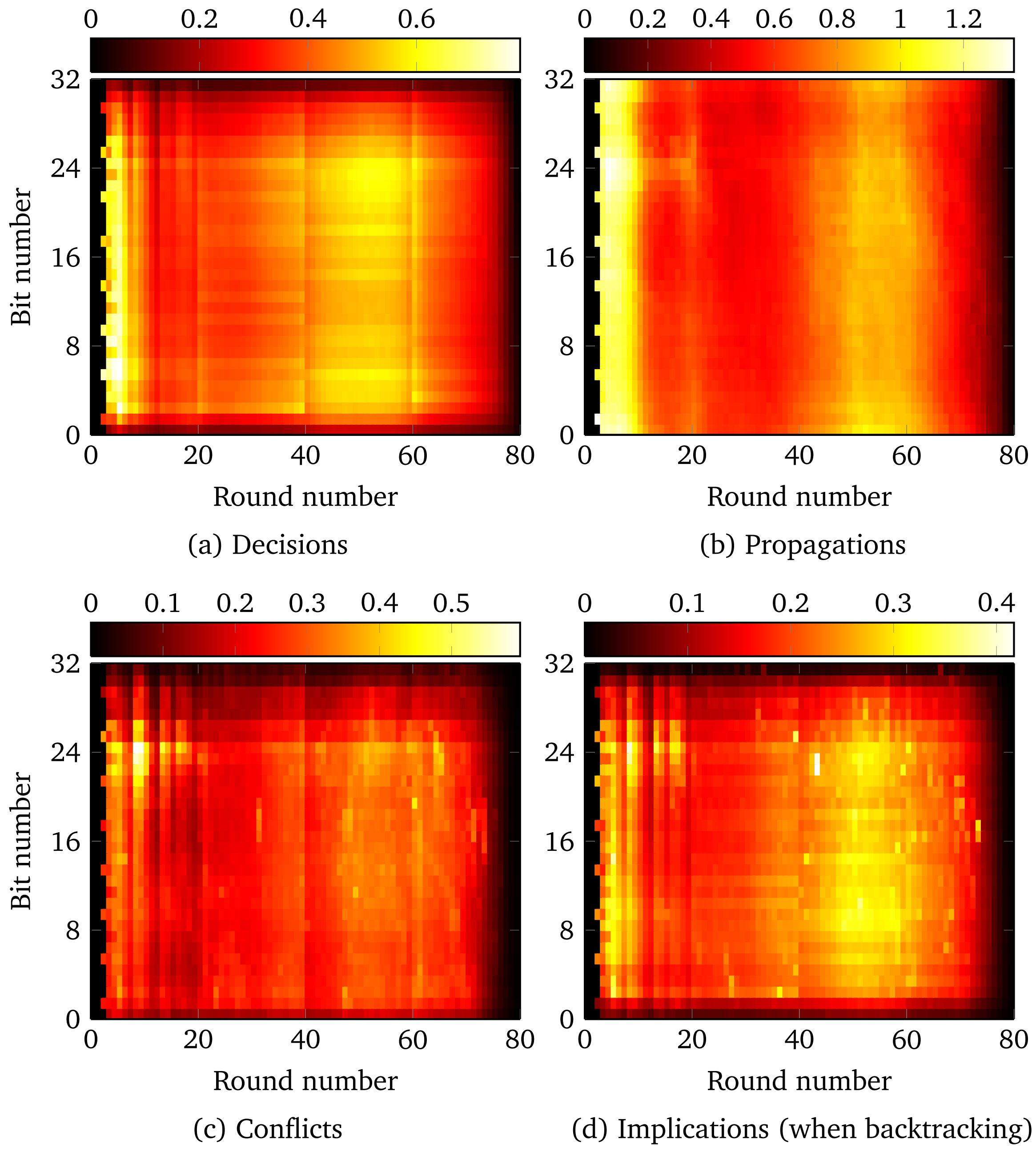


Figure A.5: Heatmaps showing how often (in parts per ten thousand) the variables encoding $f_{i,j}$ (bit j of the state combiner function for round i) participate in the given activity when using **MiniSat** and the VSIDS branching heuristic.

Table A.1: Distribution of solver events (decisions, propagations, conflicts, and implications) over instance variables. Each percentage is the mean of a sample with size $n = 100$. All numbers are accurate to within ± 1 percentage point with 95% confidence.

Variable	Decisions (%)	Propagations (%)	Conflicts (%)	Implications (%)
w	28.00	42.09	50.83	49.56
a	30.23	14.41	17.65	18.29
c_0	17.83	13.75	15.34	14.64
c_1	14.36	13.54	10.64	12.72
f	9.59	16.22	5.54	4.79

A.2 Learnt clause statistics

In experiment section 4.14, we tried to reuse the most frequently learnt clauses from several runs. If the same clause is learnt during several runs although the instances (i.e. the hash values) are different, this indicates that the clause may be valid for all message-hash pairs. Moreover, since these clauses were derived by conflict analysis, we know that they are not trivially implied by our encoding (i.e. unit propagation using the clauses of the original instance on the negation of the learnt clause will not detect a conflict). We could thus view these clauses as a defect of the encoding. (Clearly, if we had an arc-consistent encoding of SHA-1, we would also have a polynomial algorithm for finding preimages, since unit propagation would never lead to a conflict. An arc-consistent encoding might be exponential in the number of variables, however.)

In order to learn in exactly what way the encoding is deficient, we can look at the learnt clauses. As in section 4.14, we have generated and solved 100 instances, but this time with 80 rounds. We also set a timeout at 600 seconds. We will summarise some of the characteristics of the clauses which were learnt in 40 or more runs.

In Figure A.6, we show how often each variable (except variables encoding **W**) occurs in a learnt clause. The most obvious thing to notice is that variables used in rounds 0 to 19 are the most frequent, followed by variables used in rounds 40 to 59. The variables in rounds 20 to 39 and in rounds 60 to 79 are almost completely absent. Another thing to notice is that the first few lower bits and last upper bits of each word appear less frequently in learnt clauses.

For the variables encoding **W** (the message and the message schedule), shown in Figure A.7, the situation is a little bit different; there, the variables in rounds 16 to 19 appear to be even more frequent than the rest. The upper bits of each word for some rounds are also more frequent than in the other variables.

An interesting question is why the variables encoding rounds 20 to 39 and 60 to 79 are nearly absent in the learnt clauses. We know that exactly these rounds use the same round-dependent logical function, $\mathbf{Parity}(\mathbf{x},\mathbf{y},\mathbf{z}) = \mathbf{x} \oplus \mathbf{y} \oplus \mathbf{z}$, which in our encoding is encoded using $2^{4-1} = 8$ clauses of length 4. The other round-dependent logical functions, **Ch** and **Maj**, are both encoded using 6 clauses of length 3. One explanation could be that the shorter clauses propagate more easily and are therefore more likely to take part in a conflict (and contribute its literals to the final conflict clause).

We have also included Table A.2, Table A.3, and Table A.4, which list, respectively, the lengths of, the unique variables in, and the rounds of the variables in the learnt clauses.

Table A.2: Lengths of the clauses which were learnt in more than 50% of the runs.

Length	Number	Percentage (%)
4	11,246	64.78
5	2,856	16.45
6	2,800	16.13
7	459	2.64

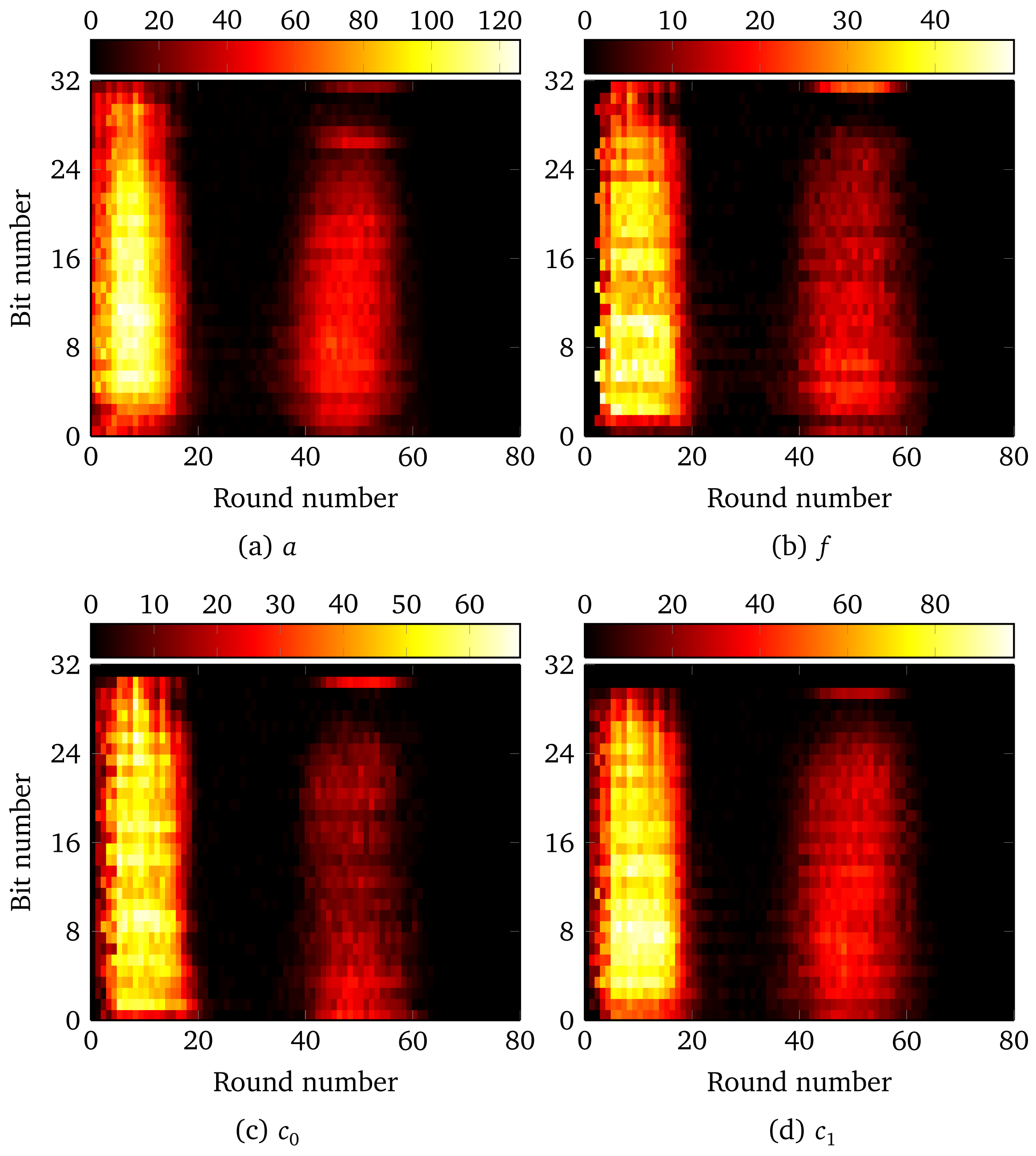


Figure A.6: Heatmaps showing how often each bit of a, f, c_0, and c_1 occur in the clauses which were learnt in more than 50% of the runs.

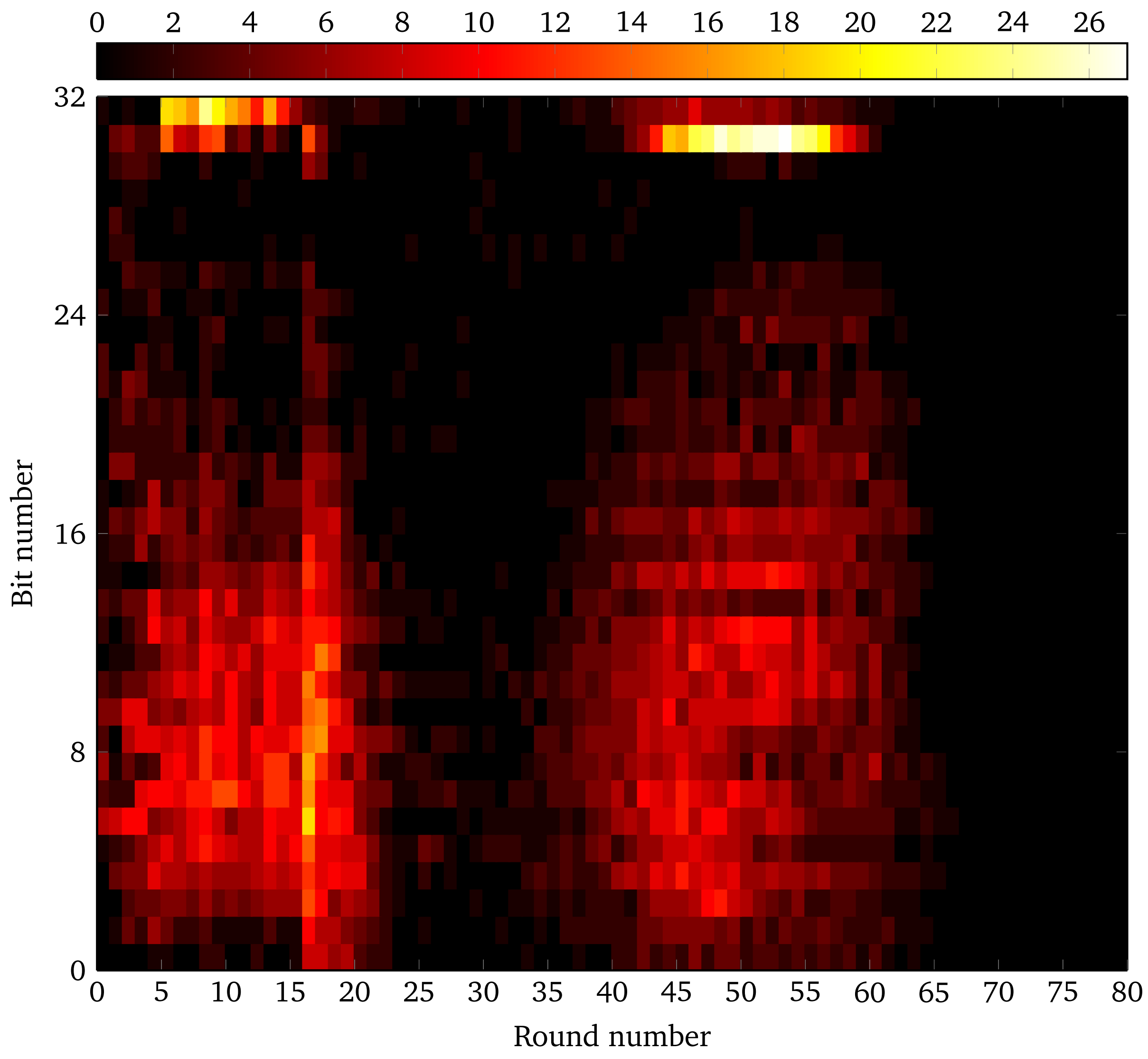


Figure A.7: Heatmaps showing how often each bit of W occur in the clauses which were learnt in more than 50% of the runs.

Table A.3: (Unique) variable types in the clauses which were learnt in more than 50% of the runs.

Variables					Number	Percentage (%)
a	c_0	c_1	f		4,830	27.82
a		c_1	f		4,210	24.25
a	c_0	c_1			3,415	19.67
a		c_1			1,841	10.60
a		c_1	f	W	989	5.70
a		c_1		W	676	3.89
	c_0	c_1	f		454	2.62
a	c_0	c_1	f	W	386	2.22
a	c_0		f		293	1.69
a	c_0	c_1		W	59	0.34
	c_0	c_1		W	53	0.31
a	c_0		f	W	32	0.18
a	c_0			W	23	0.13
		c_1	f	W	23	0.13
	c_0	c_1			20	0.12
a	c_0				19	0.11
	c_0	c_1	f	W	15	0.09
a			f	W	8	0.05
		c_1	f		8	0.05
a				W	7	0.04

Table A.4: Rounds of the variables in the clauses which were learnt in more than 50% of the runs.

Variables from rounds	Number	Percentage (%)
$t,\ t+5$	5,826	33.56
$t,\ t+1$	4,790	27.59
$t,\ t+4,\ t+5$	3,852	22.19
t	2,893	16.66

Bibliography

(1995). Secure Hash Standard. vol. 180-1, of FIPS National Institute of Standards and Technology.

(2008). ASTM-E178-08: Standard Practice for Dealing With Outlying Observations. ASTM International, West Conshohocken, PA.

Abdi, H. (2007). The Bonferonni and Šidák Corrections for Multiple Comparisons. In Encyclopedia of Measurement and Statistics pp. 103–107.

Audemard, G., Bordeaux, L., Hamadi, Y., Jabbour, S. and Sais, L. (2008). A generalized framework for conflict analysis. In Proceedings of the 11th international conference on Theory and applications of satisfiability testing SAT'08 pp. 21–27, Springer-Verlag, Berlin, Heidelberg.

Audemard, G. and Simon, L. (2009). Predicting learnt clauses quality in modern SAT solvers. In Proceedings of the 21st international jont conference on Artifical intelligence IJCAI'09 pp. 399–404, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Balint, A., Fröhlich, A., Tompkins, D. A. D. and Hoos, H. H. (2011). Sparrow2011. Technical report.

Bard, G. V. (2007). Algorithms for solving linear and polynomial systems of equations over finite fields with applications to cryptanalysis. PhD thesis, University of Maryland, College Park.

Bard, G. V. (2009). In Algebraic Cryptanalysis. Springer-Verlag Berlin Heidelberg.

Bard, G. V., Courtois, N. T. and Jefferson, C. (2007). Efficient methods for conversion and solution of sparse systems of low-degree multivariate polynomials over GF(2) via SAT-solvers. http://eprint.iacr.org/2007/024.

Béjar, R., Fernández, C. and Guitart, F. (2010). Encoding Basic Arithmetic Operations for SAT-Solvers. In Proceedings of the 2010 Conference on Artificial Intelligence Research and Development: Proceedings of the 13th International Conference of the Catalan Association for Artificial Intelligence pp. 239–248, IOS Press, Amsterdam.

Berre, D. L. and Parrain, A. (2010). The Sat4j library, release 2.2. Journal on Satisfiability, Boolean Modeling and Computation *7*, 59–64.

Biere, A. (2008). Adaptive restart control for conflict driven SAT solvers. In SAT '08: Proceedings of the 11th International Conference on Theory and Applications of Satisfiability Testing vol. 4996, of LNCS Springer-Verlag Berlin Heidelberg.

Biere, A. (2010). Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. Technical Report 10/1 Institute for Formal Models and Verification, Johannes Kepler University, Linz, Austria.

Brady, B. and Yang, Y. (2006). The Effects of Arithmetic Encodings on SAT Solver Performance. http://www.eecs.berkeley.edu/~bbrady/documents/BradyYang-report.pdf.

Brummayer, R. and Biere, A. (2006). Local two-level And-Inverter Graph minimization without blowup. In MEMICS '06: Proceedings of the 2nd Doctoral Workshop on Mathematical and Engineering Methods in Computer Science.

Buss, S. R. (1998). An introduction to proof theory. Elsevier, Amsterdam.

Chen, B. (2008). Strategies on Algebraic Attacks Using SAT Solvers. In Proceedings of the 2008 The 9th International Conference for Young Computer Scientists ICYCS '08 pp. 2204–2209, IEEE Computer Society, Washington, DC, USA.

Courtois, N. and Bard, G. V. (2007). Algebraic Cryptanalysis of the Data Encryption Standard. In Proceedings of the 11th IMA international conference on Cryptography and coding Cryptography and Coding'07 pp. 152–169,.

Courtois, N. T., Bard, G. V. and Wagner, D. (2008). Algebraic and Slide Attacks on KeeLoq. vol. 5086, of LNCS pp. 97–115,.

Courtois, N. T., O'Neil, S. and Quisquater, J.-J. (2009). Practical Algebraic Attacks on the Hitag2 Stream Cipher. vol. 5735, of LNCS pp. 167–176, Springer-Verlag Berlin Heidelberg.

D'Agostino, M. (1992). Are Tableaux an Improvement on Truth-Tables? Cut-Free proofs and Bivalence.

Davis, M., Logemann, G. and Loveland, D. (1962). A machine program for theorem-proving. Commun. ACM *5*, 394–397.

Davis, M. and Putnam, H. (1960). A Computing Procedure for Quantification Theory. J. ACM *7*, 201–215.

De, D., Kumarasubramanian, A. and Venkatesan, R. (2007). Inversion Attacks on Secure Hash Functions Using SAT Solvers. vol. 4501, of LNCS pp. 377–382,.

Dershowitz, N., Hanna, Z. and Nadel, A. (2005). A clause-based heuristic for SAT solvers. In SAT '05: Proceedings of the International Conference on Theory and Applications of Satisfiability Testing pp. 46–60, Springer-Verlag.

DiCiccio, T. J. and Efron, B. (1996). Bootstrap Confidence Intervals. Statistical Science *11*, 189–212.

Ding, J., Buchmann, J., Mohamed, M., Moahmed, W. and Weinmann, R.-P. (2008). MutantXL. In SCC ’08: Proceedings of the 1st International Conference on Symbolic Computation and Cryptography pp. 16–22,.

Dinur, I. and Shamir, A. (2008). Cube Attacks on Tweakable Black Box Polynomials. Cryptology ePrint Archive, Report 2008/385. http://eprint.iacr.org/.

Eén, N. and Biere, A. (2005). Effective preprocessing in SAT through variable and clause elimination. In SAT ’05: Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing vol. 3569, of LNCS pp. 61–75, Springer-Verlag Berlin Heidelberg.

Eén, N. and Sörensson, N. (2004). An Extensible SAT-solver. In Theory and Applications of Satisfiability Testing, (Giunchiglia, E. and Tacchella, A., eds), vol. 2919, of LNCS chapter 37, pp. 333–336. Springer-Verlag Berlin Heidelberg.

Eén, N. and Sörensson, N. (2006). Translating Pseudo-Boolean Constraints into SAT. Journal on Satisfiability, Boolean Modeling and Computation *2*, 1–26.

Eibach, T., Pilz, E. and Völkel, G. (2008). Attacking Bivium Using SAT Solvers. vol. 4996, of LNCS pp. 63–76,.

Erickson, J., Ding, J. and Christensen, C. (2009). Algebraic Cryptanalysis of SMS4: Gröbner Basis Attack and SAT Attack Compared. vol. 5984, of LNCS pp. 73–86, Springer-Verlag Berlin Heidelberg.

Faugère, J.-C. (1999). A new efficient algorithm for computing Gröbner bases (F4). Journal of Pure and Applied Algebra *139*, 61–88.

Faugère, J.-C. (2002). A new efficient algorithm for computing Gröbner bases without reduction to zero (F5). In Proceedings of the 2002 international symposium on Symbolic and algebraic computation ISSAC ’02 pp. 75–83, ACM, New York, NY, USA.

Fiorini, C., Martinelli, E. and Massacci, F. (2003). How to fake an RSA signature by encoding modular root finding as a SAT problem. Discrete Applied Mathematics *130*, 101–127.

Games, P., Keselman, H. and Clinch, J. (1979). Tests for homogeneity of variance in factorial designs. Psychological Bulletin *86*, 978–984.

Gebser, M., Kaufmann, B., Neumann, A. and Schaub, T. (2007). clasp: A conflict-driven answer set solver. In LPNMR’07 pp. 260–265, Springer.

Gelder, A. V. (2011). Careful ranking of multiple solvers with timeouts and ties. In SAT ’11: Proceedings of the 14th International Conference on Theory and Application of Satisfiability Testing vol. 6695, of LNCS pp. 317–328, Springer-Verlag Berlin Heidelberg.

Gilmore, P. (1960). A proof method for quantification theory: its justification and realization. IBM J. Res. Dev. *4*, 28–35.

Goldberg, E. and Novikov, Y. (2002). BerkMin: A fast and robust SAT-solver. In DATE '02: Proceedings of Design, Automation, and Test in Europe pp. 142–149,.

Gomes, C. P., Selman, B. and Kautz, H. (1998). Boosting combinatorial search through randomization. In AAAI '98: Proceedings of the Fifteenth National Conference on Artificial Intelligence pp. 431–437, AAAI Press.

Gwynne, M. (2010). The Interaction between Propositional Satisfiability and Applications in Cryptography and Ramsey Problems. Master's thesis Swansea University.

Gwynne, M. and Kullmann, O. (2011). Towards a better understanding of hardness. In CP '11: Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming.

Haim, S. and Heule, M. (2010). Towards Ultra Rapid Restarts. Technical report UNSW and TU Delft.

Heule, M. J. and van Maaren, H. (2006). March_dl: Adding Adaptive Heuristics and a New Branching Strategy. Journal on Satisfiability, Boolean Modeling and Computation *2*, 47–59.

Heule, M. J. H., van Zwieten, J. E., Dufour, M. and van Maaren, H. (2005). March_eq: Implementing Additional Reasoning into an Efficient Lookahead Sat Solver. In SAT 2004, (Hoos, H. H. and Mitchell, D. G., eds), vol. 3542, of Lecture Notes in Computer Science pp. 345–359, Springer.

Huang, J. (2007). The effect of restarts on the efficiency of clause learning. In Proceedings of the 20th International Joint Conference on Artifical Intelligence IJCAI'07 pp. 2318–2323, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Huber, W. A. (2012). http://stats.stackexchange.com/questions/35630/can-i-assume-log-normality-for-this-sample.

Ignatiev, A. and Semenov, A. (2011). DPLL+ROBDD Derivation Applied to Inversion of Some Cryptographic Functions. In SAT pp. 76–89,.

Jovanović, D. and Janičić, P. (2005). Logical Analysis of Hash Functions. vol. 3717, of LNAI pp. 200–215,.

Katz, J. and Lindell, Y. (2007). Chapman and Hall/CRC Press.

Laitinen, T., Junttila, T. and Niemelä, I. (2012). Conflict-driven XOR-clause learning. In Proceedings of the 15th international conference on Theory and Applications of Satisfiability Testing SAT'12 pp. 383–396, Springer-Verlag, Berlin, Heidelberg.

Land, C. E. (1972). An evaluation of approximate confidence interval estimation methods for lognormal means. Technometrics *14*, 145–158.

Luby, M., Sinclair, A. and Zuckerman, D. (1993). Optimal Speedup of Las Vegas Algorithms. Information Processing Letters *47*, 173–180.

Marques-Silva, J. P. (1999). The impact of branching heuristics in propositional satisfiability algorithms. In In 9th Portuguese Conference on Artificial Intelligence (EPIA) pp. 62–74,.

Marques-Silva, J. P. and Sakallah, K. A. (1999). GRASP: A Search Algorithm for Propositional Satisfiability. IEEE Transactions on Computers *48*, 506–521.

Massacci, F. (1999). Using Walk-SAT and Rel-SAT for Cryptographic Key Search. In IJCAI '99: Proceedings of the 16th International Joint Conference on Artificial Intelligence pp. 290–295, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Massacci, F. and Marraro, L. (1999). Towards the Formal Verification of Ciphers: Logical Cryptanalysis of DES. In Proceedings of the Third LICS Workshop on Formal Methods and Security Protocols, Federated Logic Conferences.

Massacci, F. and Marraro, L. (2000). Logical Cryptanalysis as a SAT Problem: Encoding and analysis of the U.S. data encryption standard. Journal of Automated Reasoning *24*, 165–203.

McDonald, C. (2010). Analysis of Modern Cryptographic Primitives. PhD thesis, Macquarie University.

McDonald, C., Charnes, C. and Pieprzyk, J. (2007). Attacking Bivium with MiniSat. Cryptology ePrint Archive, Report 2007/024.

McDonald, C., Hawkes, P. and Pieprzyk, J. (2009). Differential Path for SHA-1 with complexity $O(2^{52})$. Cryptology ePrint Archive, Report 2009/259.

Mironov, I. and Zhang, L. (2006). Applications of SAT Solvers to Cryptanalysis of Hash Functions. vol. 4121, of LNCS pp. 102–115,.

Morawiecki, P. and Srebrny, M. (2010). A SAT-based preimage analysis of reduced KECCAK hash functions. Cryptology ePrint Archive, Report 2010/285.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L. and Malik, S. (2001). Chaff: Engineering an Efficient SAT Solver. In Annual ACM IEEE design automation conference pp. 530–535, ACM.

Nikolić, M. (2010). Statistical Methodology for Comparison of SAT Solvers. In SAT '10: Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing pp. 209–222, Springer-Verlag New York Inc.

Pipatsrisawat, K. and Darwiche, A. (2007a). RSat 2.0: SAT Solver Description. Technical Report D–153 Automated Reasoning Group, Computer Science Department, UCLA.

Pipatsrisawat, K. and Darwiche, A. (2007b). A Lightweight Component Caching Scheme for Satisfiability Solvers. In SAT '07: Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing pp. 294–299,.

Pipatsrisawat, K. and Darwiche, A. (2009). On the power of clause-learning SAT solvers with restarts. In Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming CP'09 pp. 654–668, Springer-Verlag, Berlin, Heidelberg.

R Core Team (2012). R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing Vienna, Austria. ISBN 3-900051-07-0.

Renauld, M. and Standaert, F.-X. (2009). Algebraic Side-Channel Attacks.

Renauld, M., Standaert, F.-X. and Veyrat-Charvillon, N. (2009). Algebraic Side-Channel Attacks on the AES: Why Time also Matters in DPA. In CHES pp. 97–111,.

Rivest, R. L., Agre, B., Bailey, D. V., Crutchfield, C., Dodis, Y., Elliott, K., Khan, F. A., Krishnamurthy, J., Lin, Y., Reyzin, L., Shen, E., Sukha, J., Sutherland, D., Tromer, E. and Yin, Y. L. (2008). The MD6 hash function. A proposal to NIST for SHA-3.

Rossi, F., van Beek, P. and Walsh, T. (2006). Handbook of Constraint Programming.

Rudell, R. and Sangiovanni-Vincentelli, A. (1987). Multiple valued minimization for PLA optimization. IEEE Transactions Computer Aided Design of Integrated Circuits and Systems *6*, 727–750.

Ryan, L. (2004). Efficient algorithms for clause-learning SAT solvers. Master's thesis Simon Fraser University.

Semenov, A., Zaikin, O., Bespalov, D. and Posypkin, M. (2011a). Parallel algorithms for SAT in application to inversion problems of some discrete problems. CoRR *abs/1102.3563*.

Semenov, A., Zaikin, O., Bespalov, D. and Posypkin, M. (2011b). Parallel Logical Cryptanalysis of the Generator A5/1 in BNB-Grid System. In PaCT pp. 473–483,.

Simons, P., Niemelá, I. and Soininen, T. (2002). Extending and implementing the stable model semantics. Artif. Intell. *138*, 181–234.

Soos, M. (2010). Enhanced Gaussian Elimination in DPLL-based SAT Solvers. In Pragmatics of SAT Workshop.

Soos, M., Nohl, K. and Castelluccia, C. (2009). Extending SAT Solvers to Cryptographic Problems. vol. 5584, of LNCS pp. 244–257, Springer-Verlag Berlin Heidelberg.

Sörensson, N. and Biere, A. (2009). Minimizing Learned Clauses. In Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing SAT '09 pp. 237–243, Springer-Verlag, Berlin, Heidelberg.

Sörensson, N. and Eén, N. (2002). MiniSat v1.13 - A SAT solver with conflict-clause minimization. Technical report. SAT '05 poster.

Srebrny, M., Srebrny, M. and Stepień, L. (2007). SAT as a programming environment for linear algebra and cryptanalysis. In ISAIM.

Tompkins, D. A. D. and Hoos, H. H. (2005). UBCSAT: An Implementation and Experimentation Environment for SLS Algorithms for SAT and MAX-SAT. In Revised Selected Papers from the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004), (Hoos, H. and Mitchell, D., eds), vol. 3542, of Lecture Notes in Computer Science pp. 306–320, Springer Berlin / Heidelberg.

Tseitin, G. (1968). On the complexity of derivations in propositional calculus. In Studies in Mathematics and Mathematical Logic, Part II, (Slissenko, A. O., ed.), pp. 115–125. Consultants Bureau, New-York-London.

Vielhaber, M. (2007). Breaking ONE.FIVIUM by AIDA an Algebraic IV Differential Attack. Cryptology ePrint Archive, Report 2007/413. `http://eprint.iacr.org/`.

Vollmer, H. (1999). Introduction to Circuit Complexity. Springer Berlin.

Walsh, T. (1999). Search in a Small World. In Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence IJCAI '99 pp. 1172–1177, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Walsh, T. (2000). SAT v CSP. In CP '00: Proceedings of the 6th International Conference On Principles and Practice of Constraint Programming pp. 441–456, Springer-Verlag.

Wedler, M., Stoffel, D. and Kunz, W. (2004). Arithmetic Reasoning in DPLL-Based SAT Solving. In Proceedings of the Conference on Design, Automation and Test in Europe vol. 1, of DATE '04 IEEE Computer Society, Washington, DC, USA.

Xu, L., Hutter, F., Shen, J., Hoos, H. H. and Leyton-Brown, K. (2012). SATzilla2012: Improved Algorithm Selection Based on Cost-sensitive Classification Models. Technical report.

Zhou, X.-H., Gao, S. and Hui, S. L. (1997). Methods for Comparing the Means of Two Independent Log-Normal Samples. Biometrics *53*, 1129–1135.

Zou, G. Y., cindy Yan Huo and Taleban, J. (2009). Simple confidence intervals for lognormal means and their differences with environmental applications. Environmetrics *20*, 172–180.