SHA-1, SAT-solving, and CNF

Yusuf M Motara*, Barry V Irwin*†

*Department of Computer Science, Rhodes University, Grahamstown, South Africa

†Council for Scientific and Industrial Research (CSIR), Pretoria, South Africa

[1]y.motara@ru.ac.za

[2]b.irwin@ru.ac.za

***Abstract*—Finding a preimage for a SHA-1 hash is, at present, a computationally intractable problem. SAT-solvers have been useful tools for handling such problems and can often, through heuristics, generate acceptable solutions. This research examines the intersection between the SHA-1 preimage problem, the encoding of that problem for SAT-solving, and SAT-solving. The results demonstrate that SAT-solving is not yet a viable approach to take to solve the preimage problem, and also indicate that some of the intuitions about "good" problem encodings in the literature are likely to be incorrect.**

***Index Terms*—SAT-solving, SHA-1, preimage, CNF encoding**

The SHA-1 hash is a 160-bit cryptographic hash, standardised and promulgated by the National Institute of Standards and Technology (NIST) in 1995 [1]. It is considered to be computationally infeasible to find any input for a cryptographic hash that matches a pre-selected output; this is called the *preimage* problem. The preimage problem prespecifies the output and is therefore more difficult than the collision problem that has recently shown greater tractability [2]. However, there are many computationally infeasible problems which have proven to be sufficiently tractable when the heuristics of a modern satisfiability-solver (SAT-solver) are used. This research re-examined the preimage problem and gauged the difficulty of finding solutions using modern SAT-solvers.

The paper begins by defining the form that SHA-1 must be converted into in order to use it with a SAT-solver. Some time is spent discussing the best way to encode SHA-1 into such a form, taking into account the relevant literature on the subject. Different modern SAT-solvers are applied, and the results are presented and discussed. Lastly, conclusions are drawn about the current applicability of SAT-solvers to the preimage problem.

I. Background

This section encompasses three things: the SHA-1 hash algorithm, the encoding accepted by SAT-solvers, and a very brief overview of the field of SAT-solving.

A. SHA-1

The SHA-1 algorithm is formulated in a Merkle-Damgård [3], [4] structure that relies upon the one-wayness of the compression function for its security properties. It is convenient to study the compression function in isolation and that is precisely what has been done in this research, with the understanding that obtaining a preimage for the compression function output can be trivially converted into a preimage attack on the full hash.

The compression function itself takes place in two stages, each of which involves the creation of data to be used in one of 80 *rounds*. The first stage is called message-expansion, and involves expanding the input message of 16 32-bit words (denoted by $w_{0..15}$ in this work) into 80 32-bit words. The i^{th} bit of the word to be used in round r can be found using the following recurrence relation.

$$w_r^i = \begin{cases} w_r^i & \text{when } r < 16 \\ w_{r-3}^{i+1} \oplus w_{r-8}^{i+1} \oplus w_{r-14}^{i+1} \oplus w_{r-16}^{i+1} & \text{otherwise} \end{cases} \tag{1}$$

The usual method of calculating w-values during message-expansion relies upon previously-calculated w-values. However, note that Equation 1 is entirely linear and it is therefore possible to express the value of any particular bit purely in terms of the bits of the initial $w_{0..15}$ words. Expressing a bit in terms of w-values can be called the *bitpattern* formulation for a particular bit.

The second stage uses addition, rotation, and the majority $(b \wedge c) \vee (b \wedge d) \vee (c \wedge d)$, choice $(b \wedge c) \vee (\neg b \wedge d)$, and parity $b \oplus c \oplus d$ functions to achieve the confusion and diffusion that are necessary in a cryptographic hash. The per-round output in this stage is denoted by the symbol a, and it is therefore the last five 32-bit words, comprising the final 160 bits generated by the compression function, which will have particular values for a given 16-word input.

The output may be regarded as a single 160-bit value. Let u^0 be the most significant bit of this value, where the superscript denotes the bit position. Each bit of u is associated with a corresponding set of operations that generated the appropriate bit of an a value. Let $C(n)$ be the corresponding equation for u^n, and let $C'(n)$ be the negation of $C(n)$. Now it is true that:

$$1 = \bigwedge_{n=0}^{160} \begin{cases} C(n) & \text{if } u^n = 1 \\ C'(n) & \text{if } u^n = 0 \end{cases} \tag{2}$$

This fact can be used to arrive at a single equation which expresses the output of the SHA-1 compression function for a given input.

B. Conjunctive Normal Form

A satisfiability solver accepts a set of logical statements phrased in conjunctive normal form (CNF). This form of a formula represents an equation as a conjunction of disjunctions. Each set of disjunctions is called a *clause* or *covering*, and each variable is called a *literal*. Clauses cannot be negated, but individual literals may be. The order of clauses is irrelevant.

Example 1: CNF. Consider the function $(x_0 \wedge x_1) \vee (x_2 \vee x_3) \wedge \neg(x_2 \wedge \neg x_0)$. It can be converted to CNF by using De Morgan's law to remove all negated clauses, distributing $\vee$ operations over $\wedge$ operations so that $a \vee (b \wedge c) \mapsto (a \vee b) \wedge (a \vee c)$, and then simplifying as follows:

$$\begin{aligned}
&(x_0 \wedge x_1) \vee (x_2 \vee x_3) \wedge \neg(x_2 \wedge \neg x_0)\\
&= (x_0 \wedge x_1) \vee (x_2 \vee x_3) \wedge (\neg x_2 \vee x_0)\\
&= (x_0 \vee ((x_2 \vee x_3) \wedge (\neg x_2 \vee x_0)) \wedge (x_1 \vee ((x_2 \vee x_3) \wedge (\neg x_2 \vee x_0))\\
&= (x_0 \vee (x_2 \vee x_3)) \wedge (x_0 \vee (\neg x_2 \vee x_0)) \wedge (x_1 \vee (x_2 \vee x_3)) \wedge\\
&\quad (x_1 \vee (\neg x_2 \vee x_0))\\
&= (x_0 \vee x_2 \vee x_3) \wedge (x_0 \vee \neg x_2 \vee x_0) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee\\
&\quad \neg x_2 \vee x_0)\\
&= (x_0 \vee x_2 \vee x_3) \wedge (x_0 \vee \neg x_2) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_0)\\
&= (x_0 \vee x_2 \vee x_3) \wedge (x_0 \vee \neg x_2) \wedge (x_1 \vee x_2 \vee x_3)\\
&= (x_0 \vee (\neg x_2 \wedge x_3)) \wedge (x_1 \vee x_2 \vee x_3)\\
&= (x_0 \vee \neg x_2) \wedge (x_3 \vee x_0) \wedge (x_1 \vee x_2 \vee x_3)
\end{aligned}$$

The clauses of a minimal CNF function can be viewed as a set of "filters" which exclude 0-valued rows of the truth table. Therefore, finding a preimage of a CNF function necessarily involves evaluating each of the clauses of the function, and discarding those rows which do not match. For functions where the number of variables is large, and the number of 1-valued rows is small, this process of elimination can take a great deal of computational effort.

Converting a function to CNF can be difficult; see Example 1. Though mechanical, the process involves computationally-expensive distribution of terms and application of boolean identities, and could result in an exponential increase in the size of the resulting CNF. For example, the formula $(a \wedge b) \vee (c \wedge d) \vee (e \wedge f)$ leads to the CNF $(a \vee c \vee e) \wedge (a \vee c \vee f) \wedge (a \vee d \vee e) \wedge (a \vee d \vee f) \wedge (b \vee c \vee e) \wedge (b \vee c \vee f) \wedge (b \vee d \vee e) \wedge (b \vee d \vee f)$. Note that the 3 initial clauses have become $2^3 = 8$ clauses, each of which is more complex than the initial clauses. In fact, this expansion of n clauses to 2^n clauses is not unusual, and many parts of the SHA-1 compression function — such as calculations involving $\oplus$ — lead to an exponential expansion of the resulting CNF when $\vee$s are distributed over $\wedge$s.

However, one of the great advantages of CNF is that any function can be converted to an *equisatisfiable* CNF via Tseitin encoding [5] with, at most, a linear increase in the number of terms. An equisatisfiable formula has additional variables, but is only satisfiable whenever the original formula is satisfiable. Tseitin encoding involves replacing each operation x_0 **op** x_1 with a corresponding CNF sub-expression that uses a new variable x_2:

- $x_0 \wedge x_1 \mapsto (\neg x_0 \vee \neg x_1 \vee x_2) \wedge (x_0 \vee \neg x_2) \wedge (x_1 \vee \neg x_2)$
- $x_0 \vee x_1 \mapsto (\neg x_0 \vee \neg x_1 \vee \neg x_2) \wedge (x_0 \vee x_2) \wedge (x_1 \vee x_2)$
- $\neg x_0 \mapsto (\neg x_0 \vee \neg x_2) \wedge (x_0 \vee x_2)$
- $x_0 \oplus x_1 \mapsto (\neg x_0 \vee \neg x_1 \vee \neg x_2) \wedge (\neg x_0 \vee x_1 \vee x_2) \wedge (x_0 \vee \neg x_1 \vee x_2) \wedge (x_0 \vee x_1 \vee \neg x_2)$

The new variable takes the value of the output of the expression, and can therefore be used in further expressions as a proxy for that output. Once a solution has been obtained, the new variables can be discarded and the values of the original variables substituted into the the original equation, with the same result being obtained.

A CNF formula can be stored compactly as a vector of clauses, where each clause may have a maximum of three terms. The index of the clause in the vector identifies it uniquely. This representation makes it easy to store a large formula with minimal overhead. It also makes it simple to create files in the *de facto* standard DIMACS format, in which a simple header line is followed by a single clause per line.

Finding a preimage for a CNF formula is equivalent to finding a set of inputs which will satisfy the formula. The ease of converting boolean formulae into CNF via the Tseitin transformation has led to CNF being accepted as the *de facto* standard for use with satisfiability (SAT) solvers, which exist to solve the well-known (and NP-complete) boolean satisfiability problem [6]. Despite being theoretically unsolvable in computationally-feasible time, many practical boolean satisfiability problems can be solved heuristically using modern SAT solvers [7]. In the SAT literature, clauses are often called *constraints* since they constrain the possible solution space.

C. SAT-solving

SAT algorithms may be split into two broad categories:

DPLL-based. Davis-Putnam-Logemann-Loveland [8] (DPLL) solvers (either backtracking or non-backtracking) try to find logical contradictions between variable assignments, and thus eventually derive an implication graph which contains no contradictions. Such an implication graph is a solution. Powerful modern solvers such as MiniSat [9] are DPLL-based. Conflict-Driven Clause Learning (CDCL) solvers are advanced variants of DPLL solvers.

Stochastic Local Search. Whereas DPLL-based solvers work by proving relationships between the variables, stochastic local search (SLS) solvers — also known as "random walk" solvers — start with an assignment of random values to each variable. A limited number of variables is then flipped, and the solution which has the least number of unsatisfied clauses is chosen. Using this solution as the new assignment, the same procedure is repeated until a solution where *all* clauses are satisfied is obtained. [10, Ch. 8]

The split is, in some instances, more theoretical than practical since there is a great cross-pollination of ideas and techniques between different families of solvers; indeed, solvers such as SATzilla [11] choose between what they believe to be the most "appropriate" method for a given problem. In the case of finding preimages, it is known that the formula *is* satisfiable (SAT) instead of unsatisfiable (UNSAT). Where a solution is known to exist, SLS solvers may significantly outperform DPLL-based solvers [12], [13]. However, when applied to a hash function, SLS solvers have performed much worse than their DPLL-based counterparts [14].

A big drawback of SAT-solving solutions is that they are of limited generality. Although the basic algorithms used during DPLL-solving stay the same, some of the choices made by a solver are effectively arbitrary. After how many conflicts should a solver stop pursuing a branch of reasoning? Which variable should a solver try to find a value for? How much of an abandoned branch of reasoning might it be useful to keep for future reference? These questions are decided in different ways by different solvers (and different algorithm variants). A SAT solver, applied to a particular problem, is very sensitive to the parameters which govern its behaviour. The advantage of this is that there may be a particular set of parameters — Lingeling [15], for example, has approximately 340 tunable parameters! — which *could* find a SHA-1 preimage within a relatively short span of time. The disadvantage of such a solution is that it tells the computer scientist a great deal about the behaviour of various SAT-solving algorithms, but very little about the problem domain for which those particular parameters happened to work. It is therefore unlikely to lead to a cognitive breakthrough or generalisable understanding. Even worse, there is a possibility that a SAT solver will only find a solution for a particular problem case, and not for problems of a particular type, since "it is exactly the non-adversarial nature of practical instances that is exploited by SAT solvers" [7]. One practical consequence of this is that the time taken to solve a particular problem — even if the problem may be considered "similar" to a previously-solved problem — is very difficult to predict.

II. CNF ENCODING OF SHA-1

The most relevant works in this regard are [16]–[19]. In particular, [16] comprehensively discusses a minimal encoding of SHA-1 in order to find preimages, and conducts numerous experiments to identify the best ways to SAT-solve the resulting CNF. Most of the work was done on a reduced-round variant of SHA-1, focusing on the first 20–23 rounds. Interestingly, this range ends just before the Strict Avalanche Criterion is satisfied [20], which effectively means that there remain some correlations between bit values within the examined range.

A SAT solver uses the technique of *unit propagation* to simplify a problem. A unit clause is a clause consisting of a single literal (which may be negated). When such a clause is encountered, all clauses containing the literal may be removed: if the problem is satisfiable, then the literal will make all clauses containing it satisfiable as well. In addition to this, if a clause containing the *negation* of the literal is encountered, then the literal may be removed from this clause: it cannot possibly be true in any solution, and can therefore not contribute to the meaning of the clause.

Eén highlights the importance of *arc consistency* as a worthwhile property of an encoding [19]; an arc consistent encoding allows unit propagation to be used much more effectively, often to the extent of solving a problem entirely. A definition of arc consistency taken from [19] that is specific to CNF encoding is:

> **Definition.** Let $x = (x_1, x_2, \ldots, x_n)$ be a set of constraint variables, $t = (t_1, t_2, \ldots, t_m)$ a set of introduced variables. A satisfiability equivalent CNF translation $\varphi(x, t)$ of a constraint $\mathcal{C}(x)$ is said to be *arc-consistent* under unit propagation *iff* for every partial assignment σ, performing unit propagation on $\varphi(x, t)$ will extend the assignment to σ' such that every unbound constraint variable x_i in σ' can be bound to either True or False without the assignment becoming inconsistent with $\mathcal{C}(x)$ in either case.

More informally, a formula has constraints that must be satisfied by any solution; an equisatisfiable formula should try to represent these constraints faithfully; and an arc consistent equisatisfiable formula is one where an assignment leads unambiguously to constraint satisfaction or unsatisfiability. Thus, arc consistent representations of a formula may be efficiently solved for satisfiable solutions.

Xor constraints occur during message-expansion, during the encoding of addition, and during the parity function (used as the f-function in 40 of the 80 SHA-1 rounds). Unfortunately, they also involve the longest clauses (four 3-term clauses) and the largest number of these clauses (four, as opposed to a maximum of three to encode other operations). A naïve encoding of an xor constraint such as $p \oplus q \oplus r$ thus involves taking any two of the variables, applying the Tseitin transformation, and then using the resulting additional Tseitin variable as a participant in another Tseitin transformation; viz.

$$\begin{aligned} p \oplus q \oplus r = {} & ((\neg p \vee \neg q \vee \neg t_0) \wedge (\neg p \vee q \vee t_0) \wedge (p \vee \neg q \vee \\ & t_0) \wedge (p \vee q \vee \neg t_0)) \oplus r \\ = {} & (\neg p \vee \neg q \vee \neg t_0) \wedge (\neg p \vee q \vee t_0) \wedge (p \vee \neg q \vee t_0) \wedge \\ & (p \vee q \vee \neg t_0) \wedge (\neg t_0 \vee \neg r \vee \neg t_1) \wedge (\neg t_0 \vee r \vee \\ & t_1) \wedge (t_0 \vee \neg r \vee t_1) \wedge (t_0 \vee r \vee \neg t_1) \end{aligned}$$

Thus does a constraint involving n terms expand to 2^{n-1} 3-clause constraints. Bard [21], observing that n causes problems of scale, suggests a pre-processing of n-term xor-clauses to break them down into c-term xor-clauses, $c < n$, where each new clause introduces an additional variable; c is called the *cutting number*.

Example 2: Pre-processing xor clauses [21]. Assume that the 8-term formula $p \oplus q \oplus r \oplus s \oplus t \oplus u \oplus v \oplus w$ must be converted to CNF. As above, this would require $2^{8-1} = 128$ 3-term clauses in a naïve Tseitin encoding. If we apply Bard's procedure using $c = 4$, however, we end up with the equisatisfiable clauses $p \oplus q \oplus r \oplus s \oplus x_0$, $x_0 \oplus s \oplus t \oplus u \oplus x_1$, and $x_1 \oplus v \oplus w$. The single clause has been split into three clauses, each having a maximum of $c + 1$ terms, and the corresponding number of clauses when converted to CNF is $2^4 + 2^4 + 2^2 = 16 + 16 + 4 = 36$; 92 clauses fewer than would be obtained from a naïve encoding. The cost of each clause is the introduction of an additional "dummy" variable, denoted by x_i above. According to Bard, the optimal cutting number is 6. It is also known that the clauses-to-variables ratio of a CNF formula is correlated with the difficulty of solving that formula [22, p. 110–111]. Nossum [16] therefore attempts to reduce this ratio, encoding the addition step of a SHA-1 round by using the Espresso heuristic logic minimizer [23] to express formulae using the least number of terms; using the CryptLogVer toolkit [24] for a similar reason; and using a straight Tseitin transformation (for comparison purposes).

By comparison, Legendre [18] hand-crafts the SHA-1 CNF encoding in an attempt to decrease the complexity of solving the formula, from the perspective of a DPLL-based SAT solver. The clauses-to-variables ratio is ignored in favour of simplifications and creating "logical bridges" [18, p. 16] — clauses containing only two variables — that may help during solving. Unfortunately, while improved results for the MD5 algorithm are demonstrated, improved results for SHA-1 are absent. The importance of this approach in the context of SHA-1 is therefore uncertain.

TABLE I
CNF ENCODINGS, 80 ROUNDS OF SHA-1

Ref.	Encoding	Clauses	Variables	Ratio	2-clauses
[16]	Espresso	478,476	13,408	35.69	unknown
[16]	CryptLogVer	248,220	44,812	5.54	unknown
[16]	Simple	223,551	56,108	3.98	unknown
[18]	Hand-crafted	491,791	12,779	38.48	259
[18]	", simplified	375,195	12,771	29.38	908

Both [18] and [16] call out the addition step as being worthy of special effort when encoding. [16] try several approaches: using the Espresso heuristic logic minimizer [23] to express formulae using the least number of terms; using the CryptLogVer toolkit [24] for a similar reason; and using a straight Tseitin transformation. The clauses, variables, and ratios for various 80-round encodings are presented as Table I.

TABLE II
THE EFFECT OF GLUCOSE 4.0 SIMPLIFICATION OPTIONS ON A 210,121-VARIABLE, 629,597-CLAUSE CNF ENCODING OF SHA-1

Simplification options	Clauses	Variables	2-clauses
`-elim`	356,395	66,457	81,446
`-elim -asymm`	350,673	64,901	85,185
`-elim -grow=50 -asymm`	482,577	23,653	16,476
`-elim -grow=100 -asymm`	486,735	16,730	5,687
`-elim -grow=200 -asymm`	470,176	12,700	1,314
`-elim -grow=500 -asymm`	506,669	11,587	751
`-elim -grow=10000 -asymm`	2,486,170	9,277	534

No such encoding optimisations have been enacted in this work. A simple, naïve encoding of the SHA-1 algorithm, as described by Equation 2, results in 210,121 variables and 629,597 clauses, giving a clause-to-variable ratio of 2.996 — which is a "better" ratio than any listed in Table I. [18] focuses on longer clauses with more 2-clause bridges, giving a higher ratio. When this naïve encoding is passed to the Glucose 4.0 solver[1] [9], [25] along with suitable simplification options, the result is a CNF encoding with 470,176 clauses, 12,700 variables, and 1,314 2-clauses — arguably a "better" encoding than the hand-crafted one, under the assumption that simplification works towards making solving easier. Table II shows the effect of using various simplification options on the simple CNF encoding that has already been described.

TABLE III
THE EFFECT OF GLUCOSE 4.0 SIMPLIFICATION OPTIONS ON A 267,603-VARIABLE, 857,477-CLAUSE CNF ENCODING OF SHA-1

Simplification options	Clauses	Variables	2-clauses
`-elim`	582,924	98,119	81,093
`-elim -asymm`	577,616	96,653	85,542
`-elim -grow=50 -asymm`	983,082	38,184	16,690
`-elim -grow=100 -asymm`	1,185,483	28,789	5,323
`-elim -grow=200 -asymm`	1,186,745	24,930	1,247
`-elim -grow=500 -asymm`	2,304,868	20,423	659
`-elim -grow=10000 -asymm`	14,422,346	14,835	448

For comparison purposes, Table III presents figures for an encoding of SHA-1 that uses bitpatterns. The figures demonstrate that the trends caused by simplification are similar, even when the initial encoding of the problem is quite different: the number of variables decreases, the number of clauses increases, and the number of 2-clauses decreases. The 2-clauses column, in particular, is interesting because of how similar it is between the encodings.

Not directly evident from Tables II and III, but worthwhile to note, is the time and space costs of different initial encodings. More effort spent on simplification requires more computational power, and the final rows of Tables II and III took 7 minutes and 120 minutes respectively, and resulted in simplified DIMACS files that were 113Mb and 858Mb respectively.

No encoding tricks, such as those already described, have been used to simplify the CNF encoding in this work. Taking into consideration the previous work on this topic, the experimental results detailed above, and the possible advantages and disadvantages, the conclusion reached is that the built-in simplification routines used by modern solvers (see, for example, [26]) are likely to be powerful enough for all practical purposes; there is little to be gained by hand-tweaking a SHA-1 encoding.

There are basic encodings which appear to be objectively worse to begin from; a comparative examination of Tables II and III appears to show that the bitpattern w-formulation leads to one such encoding. However, the results of the following section will demonstrate that this appearance is deceptive.

III. SAT-SOLVING

The Glucose [9], [25], YalSAT[2] [27], Plingeling[3] [15], and CryptoMiniSat[4] [28] SAT solvers were chosen as being representative of a cross-section of SAT-solving approaches. Glucose is a modern, state-of-the-art CDCL solver; YalSAT

[1]Glucose version 4.0, with Glucose Syrup

[2]YalSAT version 03l

[3]Plingeling version bbc-9239380-160707

[4]CryptoMiniSat version 5.0.0 (with gaussian elimination)

is a modern take on a Stochastic Local Search solver; Plingeling is a SAT solver that attempts to exploit multi-core architectures; and CryptoMiniSat is a well-regarded open-source CDCL solver, originally targeted towards solving cryptographic problems, which supports a xor-clause extension to the DIMACS format. All solvers were run in their default configurations and solvers which did not find a solution within 10 minutes were terminated.

TABLE IV
SAT-SOLVING TO FIND A PREIMAGE

Input bits	Solver	Time taken (s)	
		standard	bitpattern
6	Glucose	1.3	1.3
	YalSAT / ProbSat / CSCCSat14	–	–
	Plingeling	1.0	1.4
	CryptoMiniSat	2.5	2.2
12	Glucose	16.4	7.1
	YalSAT / ProbSat / CSCCSat14	–	–
	Plingeling	15.9	7.6
	CryptoMiniSat	3.6	16.6
16	Glucose	135.6	252.0
	YalSAT / ProbSat / CSCCSat14	–	–
	Plingeling	25.8	250.0
	CryptoMiniSat	256.1	18.5
18	Glucose	403.6	138.2
	YalSAT / ProbSat / CSCCSat14	–	–
	Plingeling	82.7	463.2
	CryptoMiniSat	–	181.9
20	Glucose	227.8	132.0
	YalSAT / ProbSat / CSCCSat14	–	–
	Plingeling	–	–
	CryptoMiniSat	–	–
22	Glucose	–	–
	YalSAT / ProbSat / CSCCSat14	–	–
	Plingeling	–	–
	CryptoMiniSat	–	–

For each run, a particular number of bits were allowed to vary and the remaining bits were set to fixed values. This made it possible to examine the behaviour of the SAT-solvers as the search space increased.

SAT solver results are presented in Table IV. YalSAT, in line with the reported results of [14], was unable to solve any of the problems. To check whether this was a problem with YalSAT or with the SLS approach, two other independently-developed SLS solvers (ProbSat [29] and CSCCSat14 [30], [31]) were applied to the smallest (6-bit input) problem. Both failed to arrive at a solution within 10 minutes, and this confirms that the problem is likely to be the SLS approach rather than the YalSAT solver itself. CDCL-based solvers worked somewhat better, with Glucose outperforming both Plingeling and CryptoMiniSat. The time taken to find solutions varied significantly and unpredictably, and neither the standard w-encoding nor the bitpattern w-encoding showed itself to be definitively superior. Glucose seemed to find the bitpattern w-encoding to be easier to solve for bit-lengths 18 and 20, but more difficult for bit-length 16; Plingeling found the bitpattern w-encoding to be uniformly harder to work with; and CryptoMiniSat seemed to find the bitpattern w-encoding to be uniformly easier to work with. This unpredictability casts some doubt on the weight that should be given to "second-guessing" a solver by simplifying or hand-tweaking a CNF encoding.

IV. Discussion & Conclusions

No solver could find a preimage for more than 20 bits of input. With that being said, finding a preimage for hash inputs of ≤ 20 bits on consumer-level hardware is no mean feat; if anything, it demonstrates the enormous advances that have been made in the field of SAT-solving over the past decades. Another two decades of progress in the field may make SAT-solving for larger bit-lengths much easier. It would currently be faster to exhaustively check all n-bit inputs than it would be to run a SAT-solver for n bits, and this means that SAT-solvers are worse than brute force solutions as far as the preimage problem is concerned.

This result is not surprising. However, the vastly different behaviours of the different SAT-solvers when faced with different encodings *is* surprising. Much research, mentioned in this work, has tried to find the "best" encoding for SAT-solving and these results appear to show that that effort is misdirected: additional 2-clauses, fewer clauses, and clause length have vastly different effects, depending on the SAT-solver being employed. The relationship between SAT-solver, heuristics, problem, and encoding is complex and deserving of further study.

Bellare [32] has demonstrated that the SHA-1 hash possesses an admirable amount of *balance*: that is, the output of the compression function contains approximately the same number of ones and zeroes. Previous research undertaken by the authors analyzed the strict avalanche criterion in relation to SHA-1 [20] and found that there was no statistical correlation whatsoever between input and output bits, starting from very early in the compression function. Taken together, these two results would imply that a solver which is able to find preimages for inputs of ≈ 160 bits would also be able to find a preimage for most SHA-1 hashes.

Acknowledgement

This work was undertaken in the Distributed Multimedia CoE at Rhodes University, with financial support from Telkom SA, Tellabs, Easttel, Bright Ideas 39, THRIP and NRF SA (UID 75107). The authors acknowledge that opinions, findings and conclusions or recommendations expressed here are those of the author(s) and that none of the above mentioned sponsors accept liability whatsoever in this regard.

References

[1] NIST, "Federal information processing standard (FIPS) 180-1. Secure hash standard," *National Institute of Standards and Technology*, vol. 17, 1995.

[2] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov, "The first collision for full sha-1," Cryptology ePrint Archive, Report 2017/190, 2017, preprint; http://eprint.iacr.org/2017/190.

[3] R. C. Merkle, "Secrecy, authentication, and public key systems," Ph.D. dissertation, Department of Electrical Engineering, Stanford University, 1979.

[4] I. B. Damgård, “A design principle for hash functions,” in *Conference on the Theory and Application of Cryptology*. Springer, 1989, pp. 416–427.

[5] G. S. Tseitin, “On the complexity of derivation in propositional calculus,” in *Automation of reasoning*. Springer, 1983, pp. 466–483.

[6] S. A. Cook, “The complexity of theorem-proving procedures,” in *Proceedings of the Third Annual ACM Symposium on Theory of computing*. ACM, 1971, pp. 151–158.

[7] S. Malik and L. Zhang, “Boolean satisfiability from theoretical hardness to practical success,” *Communications of the ACM*, vol. 52, no. 8, pp. 76–82, 2009.

[8] M. Davis, G. Logemann, and D. Loveland, “A machine program for theorem-proving,” *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, 1962.

[9] N. Eén and N. Sörensson, “An extensible SAT-solver,” in *Theory and applications of satisfiability testing*. Springer, 2003, pp. 502–518.

[10] A. Biere, M. Heule, and H. van Maaren, *Handbook of satisfiability*, ser. Frontiers in Artificial Intelligence and Applications. IOS press, 2009, vol. 185.

[11] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, “SATzilla: portfolio-based algorithm selection for SAT,” *Journal of Artificial Intelligence Research*, pp. 565–606, 2008.

[12] H. Kautz and B. Selman, “Pushing the envelope: planning, propositional logic, and stochastic search,” in *Proceedings of the National Conference on Artificial Intelligence*, 1996, pp. 1194–1201.

[13] B. Selman, H. Kautz, B. Cohen *et al.*, “Local search strategies for satisfiability testing,” *Cliques, coloring, and satisfiability: Second DIMACS implementation challenge*, vol. 26, pp. 521–532, 1993.

[14] F. Massacci, “Using Walk-SAT and Rel-Sat for Cryptographic Key Search,” in *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*. Morgan Kaufmann Publishers Inc., 1999, pp. 290–295.

[15] A. Biere, “Lingeling, Plingeling and Treengeling Entering the SAT Competition 2013,” *Proceedings of SAT Competition 2013*, pp. 51–52, 2013.

[16] V. Nossum, “SAT-based preimage attacks on SHA-1,” Master’s thesis, Department of Informatics, University of Oslo, Norway, 2012.

[17] F. Legendre, G. Dequen, and M. Krajecki, “Encoding hash functions as a SAT problem,” in *24th International Conference on Tools with Artificial Intelligence (ICTAI 2012)*. IEEE, 2012, pp. 916–921.

[18] ——, “Logical Reasoning to Detect Weaknesses About SHA-1 and MD4/5,” *IACR Cryptology ePrint Archive*, vol. 2014, p. 239, 2014.

[19] N. Eén and N. Sörensson, “Translating pseudo-boolean constraints into SAT,” *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, pp. 1–26, 2006.

[20] Y. M. Motara and B. V. Irwin, “SHA-1 and the Strict Avalanche Criterion,” in *Proceedings of the 2016 Information Security for South Africa (ISSA 2016) Conference*. IEEE, 2016.

[21] G. V. Bard, “Algorithms for solving linear and polynomial systems of equations over finite fields, with applications to cryptanalysis,” Ph.D. dissertation, Department of Mathematics, University of Maryland, United States of America, 2007.

[22] F. Van Harmelen, V. Lifschitz, and B. Porter, *Handbook of knowledge representation*. Elsevier, 2008, vol. 1.

[23] R. L. Rudell and A. Sangiovanni-Vincentelli, “Multiple-valued minimization for PLA optimization,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 6, no. 5, pp. 727–750, 1987.

[24] P. Morawiecki and M. Srebrny, “A SAT-based preimage analysis of reduced KECCAK hash functions,” *Information Processing Letters*, vol. 113, no. 10, pp. 392–397, 2013.

[25] G. Audemard and L. Simon, “Predicting learnt clauses quality in modern SAT solvers,” in *Proceedings of the 21st international joint conference on Artifical Intelligence*. Morgan Kaufmann Publishers Inc., 2009, pp. 399–404.

[26] N. Eén and A. Biere, “Effective preprocessing in SAT through variable and clause elimination,” in *International conference on theory and applications of satisfiability testing*. Springer, 2005, pp. 61–75.

[27] A. Biere, “Yet Another Local Search Solver and Lingeling and friends entering the SAT competition 2014,” *Proceedings of SAT Competition 2014*, pp. 39–40, 2014.

[28] M. Soos and M. Lindauer, “The CryptoMiniSat-4.4 set of solvers at the SAT Race 2015,” *SAT Race*, 2015.

[29] A. Balint and U. Schöning, “Choosing probability distributions for stochastic local search and the role of make versus break,” in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2012, pp. 16–29.

[30] C. Luo, S. Cai, W. Wu, and K. Su, “Focused random walk with configuration checking and break minimum for satisfiability,” in *International Conference on Principles and Practice of Constraint Programming*. Springer, 2013, pp. 481–496.

[31] ——, “Double Configuration Checking in Stochastic Local Search for Satisfiability,” in *Proceedings of the 28th AAAI Conference on Artificial Intelligence*, 2014, pp. 2703–2709.

[32] M. Bellare and T. Kohno, “Hash function balance and its impact on birthday attacks,” in *Advances in Cryptology (EUROCRYPT 2004)*. Springer, 2004, pp. 401–418.

Yusuf M Motara is a lecturer at Rhodes University and a member of the Security and Networks Research Group (SNRG) at that institution. His interests are information security, programming languages, software development, and the intersection of these areas.