Evaluation of SAT-based preimage attack optimizations

Ladislav Pápay*

Supervisor: Martin Stanek†

Katedra informatiky, FMFI UK, Mlynská Dolina 842 48 Bratislava

Abstract: SAT solvers are a universal tool for finding solutions to boolean satisfiability problems. In the past they have been used for cryptographic problems, such as finding preimages for hash functions or obtaining the key for stream ciphers. However these solutions are not easily reusable or modifiable.

In our work we create a modeling library that allows simple creation of SAT instances. We specifically focus on various cryptographic problems, however the library is generic enough that it can be used for other purposes as well.

Using this library we create models for several cryptographic hash functions. Various SAT solvers, optimizations and heuristics are evaluated on these models to compare their performance. These include the use of the *Espresso* logic minimizer to reduce the instance size, forcing custom variable branching order with help of modified SAT solvers and others.

Keywords: SAT, cryptography, hash functions, heuristics

1 Introduction

1.1 SAT solvers

SAT solvers (shortened from *satisfiability*) are programs which take a boolean satisfiability problem and a find solution or solutions to this problem. The problem, which we will call an *instance*, consists of a number of variables and boolean clauses composed of these variables. The solution, if it exists, is an assignment of truth values to the variables such that the clauses are satisfied. If no such assignment exists the instance is called *unsatisfiable*.

Most modern SAT solvers expect input in *conjunctive normal form* in which all clauses are disjunctions of literals (literal is either a variable or a negation of a variable, and at least one literal must be true) and the instance is a conjunction of such clauses (all clauses must be true). As the name suggests this is a normal form, which means every boolean expression can be converted to an equivalent one that satisfies these requirements. Thus we will from now on only consider such instances without loss of generality.

By modeling a decision problem as a SAT instance we can take advantage of the advanced optimizations and heuristics employed by modern SAT solvers. Examples of such uses in practice include formal program verification, model checking or even routing of connections in microchips.

1.1.1 Interfacing with SAT solvers

Since our work depends on providing a suitable input to SAT solvers we will briefly describe the widely supported *DIMACS*[1] *CNF* format which we use. Later in section 3.2.1 we will describe augmenting this format for optimization purposes.

The file begins with a line in form

$$\texttt{p cnf}\ \mathit{vars}\ \mathit{clauses}$$

where *vars* and *clauses* are the number of variables and clauses in this instance, respectively.

The following lines are in the form

$$v_1 \ldots v_i\ \texttt{0}$$

and each represents a single clause consisting of a disjunction of literals v_1 through v_i. Each literal v_i is an integer between 1 and *vars* for a variable in positive polarity or an integer between -1 and $-vars$ for negative polarity.

The output for a satisfiable instance is a list of *vars* numbers. For every $1 \leq v_i \leq vars$ either v_i or $-v_i$ is present in the list, indicating the polarity of the i-th variable in some satisfying assignment.

1.2 Cryptographic problems

We make use of SAT solvers to find solutions to instances which represent a cryptographic primitive. The variables represent the inputs, outputs and the internal state of the primitive. The clauses describe the behavior of the primitive as a relation between the variables. Solutions to such instances are pairs of inputs and corresponding outputs.

*lacop@lacop.net

†stanek@dcs.fmph.uniba.sk

[1] As in the case of *JPEG* the acronym refers to the institute that created the format – the *Center for Discrete Mathematics and Theoretical Computer Science.*

By placing additional restrictions on the possible solutions (by introducing more clauses) we can find a truth assignment that gives us the desired inputs and outputs to the cryptographic primitive. With a cipher this can mean finding the secret key (input) by restricting the output to observed keystream. With hash functions this can mean reversing the output for a *preimage* attack or finding two different input with the same output (a *collision*).

In our work we will focus specifically on hash functions, which we describe in more detail in the following section.

1.2.1 Hash functions

A *hash function* is designed to reduce a long input *message* into a shorter, fixed-length *digest* (short from *message digest*, also called *hash*). The computation in this direction is often designed to be fast and efficient. However, a *cryptographic* hash function should further have the property that it is infeasible to compute it in the opposite direction – that is, given a digest to find a message.[2]

By infeasible (or hard) we mean that the computation should take time exponential to some parameters of the hash function and its input, and thus is not practical for sufficient input sizes.

Formally, hash function h is a function $h : X \to Y$ where X is the (potentially unbounded) set of input messages and $h(x) \in Y = \{0,1\}^n$ is an n-bit message digest. The properties which we require these functions to have are:

Preimage resistance: Given a digest d it is hard to find a message m such that $h(m) = d$.

Second preimage resistance: Given a message m_1 it is hard to find a message $m_2 \neq m_1$ such that $h(m_1) = h(m_2)$.

Collision resistance: It is hard to find messages m_1, m_2 such that $h(m_1) = h(m_2)$ and $m_1 \neq m_2$.

Most modern hash functions have multiple rounds in which the state is modified using a *round function*. The initial state is based on the input message (with some form of padding). The final state is then used to produce the digest.

We will briefly describe two hash functions on which we focus in our work and to whose internals we will be referring later.

SHA-1, short for *Secure Hash Algorithm* is a widely used hash function based on the *Merkle-Damgård* construction [Merkle et al., 1979] with output digest size of 160 bits. The internal state consists of five 32-bit words which are updated in 80 rounds.

SHA-3 is a recently standardized hash function based on the *Keccak* family that won the selection competition against fifty other candidates. It is based on a *sponge* construction which allows the output digest size to be variable.

We will specifically focus on the variant *SHA-3-512* which has 1600-bit state updated in 24 rounds (divided into five steps called θ, ρ, π, χ and ι) and produces a 512-bit digest. The state is represented as a 5×5 matrix S of 64-bit words.

1.3 Encoding

The most important step when using SAT solvers for any problem is the encoding of the model to a suitable instance. In case of cryptographic problems this previously (see section 1.4) required significant effort of either generating the SAT instance by hand, or by first translating the model to a more suitable form and then using existing tools to generate the instance.

In our work we created a library [Pápay, 2016] which automates large parts of this process. This makes it easier, faster and less error-prone. To achieve this we made use of *boolean circuit* representation to translate almost unmodified implementations of cryptographic primitives to SAT instances transparently.

1.3.1 Boolean circuits

The cryptographic primitive we want to encode into a SAT instance can be thought of as a *boolean circuit* – a *DAG* (directed acyclic graph) in which vertices are boolean operators (*gates*, such as the *AND gate*, *XOR gate* and so on). The edges represent flow of values from one gate to another. *Inputs* are special vertices that have no incoming edges. *Outputs* are also special vertices that have exactly one incoming edge and no outgoing edges.

Given this representation the simplest way to encode this as a CNF formula would be to take each output vertex and recursively expand it in the following way: At first we start with an output vertex v encoded as (v). This will have one incoming edge from

[2] Not *the* message, since there might be multiple input messages that have the same digest.

a gate vertex g with inputs $x_1, \ldots, x_k$. We will encode this as $g(x_1, \ldots, x_k)$ where g is the appropriate boolean operation of the gate – for example, if the output vertex is connected to an *AND* gate this would be $(x_1 \wedge \ldots \wedge x_k)$.

Now we repeat this process recursively, expanding each gate node until the encoding only refers to the input vertices. The resulting encoding has to be converted into conjunctive normal form and can then be solved with any SAT solver.

However, this approach produces very large output formula with many clauses. Since we are recursively expanding each vertex until we reach the input vertices, large subgraphs which are referenced (connected by an outgoing edge) multiple times will needlessly be repeated in the output encoding. The total length of the formula then can be exponential in the size of the input circuit.

1.3.2 Tseitin transformation

To reduce the number of clauses of the resulting encoding we can use the *Tseitin transformation* [Tseitin, 1983]. Instead of generating a large number of clauses when expanding the circuit we will add a new variable for each vertex.

Each gate vertex can then be encoded as a boolean function with constant number of clauses using only variables corresponding to gates (or inputs) that are connected via an incoming edge. The following table shows how to encode the most common boolean gates with two inputs A and B. Variable C refers to the new variable representing this gate in the encoding.

AND	$A \wedge B$	$(C \vee \overline{A} \vee \overline{B}) \wedge (\overline{C} \vee A) \wedge (\overline{C} \vee B)$
OR	$A \vee B$	$(\overline{C} \vee A \vee B) \wedge (C \vee \overline{A}) \wedge (C \vee \overline{B})$
XOR	$A \oplus B$	$(\overline{C} \vee \overline{A} \vee \overline{B}) \wedge (\overline{C} \vee A \vee B) \wedge$ $(C \vee \overline{A} \vee B) \wedge (C \vee A \vee \overline{B})$

Other binary gates such as *NAND* can be encoded similarly. In case of multiple inputs we can either extend this encoding or replace the n-ary gates with multiple binary ones.

This encoding produces formula with linear number of variables and linear length with respect to the size of the boolean circuit, if we limit the it to unary and binary gates.

1.3.3 Arithmetic gates

Most cryptographic algorithms make heavy use of arithmetic operations, such as modular addition. These usually work on variables of some fixed size, for example 32-bit integers.

The n-bit variable can be represented using n binary variables in the SAT instance. However, these operations can not be performed on individual bits like in the case of boolean operators. We need to introduce additional helper variables that will represent the carry bits during the addition operation.

This can be thought of as taking the boolean circuit for a binary adder with carry (either ripple-carry or lookahead-carry), composed of several full-adder circuits. These can then be encoded using the Tseitin transformation.

However, for simplicity we extend our model of boolean circuits to support modular addition nodes directly. When encoding these nodes to CNF we use the additional carry variables and output the clauses required to model a binary adder.

1.4 Related work

The idea of using SAT solvers for cryptographic problems was first introduced in [Massacci and Marraro, 2000]. The authors designed an encoding of the *DES* (*Data Encryption Standard*) symmetric cipher and created a program to generate instances. However the work is very specific to *DES* and modifying the tool for a different cipher would require significant changes.

The first attempt to simplify this process can be found in [Jovanovic and Janicic, 2005] where *operator overloading* (see section 2.2.1) in the *C++* language is used. Our library also makes use of this feature with the additional advantage that we use a *dynamic* programming language (*Python*) and therefore require smaller changes to existing code. Additionally their work uses only simple Tseitin transform without additional optimizations, and the code for the tool is not available.

Another work which automates the generation of instances makes use of the *Verilog* hardware description language[3] [Morawiecki and Srebrny, 2013]. After writing the cryptographic primitive in this language a free, but proprietary, compiler is used to generate equations which are then turned to a CNF instance. Since *Verilog* is quite a niche language most cryptographic primitives do not have implementations available. Additionally the toolkit itself is also

[3] HDLs describe the behavior of logic circuits. They are used for programming FPGAs or designing ASICs.

not publicly available.

An in-depth analysis of *SHA-1* preimage attacks was done in [Nossum, 2012]. The instances were generated using a custom, hand-built 1000-line *C++* program. The experiments investigated the speed of various SAT solvers, effects of preprocessing and simplifying the instance and various heuristics.

2 Modeling library

The works mentioned in previous section created their models for various cryptographic problems mostly by hand. While the results obtained are interesting, they are hard to reproduce by others. Also this approach does not help to solve similar problems (like using a different hash function instead), as a new model would have to be created from scratch.

To address these issues we provide an easy to use and reusable library for modeling SAT instances. While the library can be used for modeling any problem we specifically focus on making modeling cryptographic problems as simple as possible. In this section we will state our goals for this library, describe its design and inner functionality. We will also show examples of its use.

2.1 Goals

The main goals of our library are as follows:

Existing implementation reuse: In order to simplify the modeling of cryptographic primitives as much as possible we want to allow reuse of existing implementations. Most commonly used primitives – such as hash functions, block and stream ciphers and others – have widely available implementations in all popular programming languages.

The library should therefore allow using these implementations with only minor changes. In addition to saving time this also makes the modeling less error-prone as we can build upon a well tested implementation.

Output abstraction: The library should take care of generating the output in proper format for some SAT solver. With solvers that support advanced features, such as *XOR* clauses, it should be possible to take advantage of them.

Model parsing: After successfully solving the instance with a SAT solver we obtain a model in form of a satisfying variable assignment. The library should be able to load this model and map the truth assignment back to variables defined by the user. This makes it easy to extract for example the colliding messages out of the model.

2.2 Our approach

To achieve the goals stated in previous section we take advantage of a technique called *operator overloading*. This is a feature present in many programming languages. For our library we have decided to use the *Python* language which also supports it. Python has the additional advantage that it is very popular and therefore has implementations of virtually all commonly used cryptographic primitives.

2.2.1 Operator overloading

As the name suggests, operator overloading allows us to overload (override) existing behavior of operators in some programming language. While the feature is only *syntactic sugar* (which means it does not allow us to do anything more than would be possible without it; it just simplifies the syntax) it not only greatly increases the readability of the code, but also allows us to reuse existing implementations as we will show.

The reason this feature is useful in our library is that we can change the type of some variables in existing code without having to change anything else. For example, we can take an implementation of some hash function and change the type of all variables from the built-in integers to our new data type. Without operator overloading this could would not be possible, since operators such as addition or bit shift would not be defined for our new type by the language. With operator overloading however we can provide these definitions ourselves.

Our library provides a new data type *BitVector* that supports all the operations as the built-in integer type. Since Python uses *dynamic typing* it is sufficient to change the types of the constants used by the cryptographic primitive implementation. All other variables are a result of operations on these constants and will therefore have the proper type.

2.2.2 Boolean circuit creation

The difference between the built-in integer type and *BitVector* is that while the integer variable only holds

one given value, our type instead stores how its value can be obtained from other variables.

More specifically, each time an operation is applied to one or more operands (variables of type *BitVector* or constants) the result is another instance of type *BitVector* which stores these operands. That means that the output of some cryptographic primitive is not a single value but instead a boolean circuit representation. The circuit will form a directed acyclic graph.

2.2.3 Instance generation

Once we have the boolean circuit for a model we can then take this representation and output a SAT instance using the Tseitin transformation described in section 1.3.2.

The order of clauses in the output is irrelevant, however since the circuit forms a DAG we can process it in topological order. For every node (representing an operator applied to one or more operands) we first assign numbers to all required variables. This is because the *DIMACS* input format (described in section 1.1.1) used by most SAT solvers uses integers to refer to variables.

In most cases we have one variable for every bit of the vector. However, an addition node needs additional carry variables. On the other hand, for a negation node we don't need to introduce additional variables, as we can simply use the variables already assigned to the operand with reversed polarity. Similarly for cyclic bit shift we can reuse existing variables but with different ordering.

Once all variables have been assigned integer values we pass through all the nodes again. This time we generate the clauses which model the operator behavior. The number of clauses required depends on the operator and the bit width of the *BitVector* node.

2.2.4 Solving and working with solution

After the model has been turned into an instance we can run a SAT solver on this generated list of clauses and wait for it to terminate. If the instance is satisfiable the solver will find a satisfying truth assignment and output it as truth values for all the variables. The library will load, parse and store this data for later use.

We can then easily query any of the *BitVector* variables for its value. The mapping from variables to integer labels will be used to find the appropriate assignment and to reconstruct the value of the node.

3 Optimizations

Modern SAT solvers use many advanced optimizations and heuristics to improve the solving time. However these are usually designed for general problem instances. With additional knowledge about a specific problem we can try to come up with better rules which lead to decrease of solving time.

In this section we will describe several optimizations that we implemented and evaluated. Some of these involve a simple change in output instance generation, others involve changes to the SAT solvers themselves.

3.1 Expression encoding

The automatic and transparent conversion from unmodified source code to SAT instance via operator overloading and boolean circuits may lead to suboptimal encoding of various expressions. For example, the *choice* round function in *SHA-1*

$$Ch(x,y,z) = (x \wedge y) \oplus (\overline{x} \wedge z)$$

leads to an encoding with three variables and ten clauses (three clauses for each *and* gate and four for the *xor* gate) if we apply the Tseitin transformation directly. A better encoding using just six clauses and one variable is shown in [Nossum, 2012]. In fact, it is possible to encode this expression with just four clauses.

To solve this issue we provide an expression optimization function in our library. Any n-ary expression can be wrapped using this function to replace the standard Tseitin encoding with a (potentially) smaller one, in terms of number of extra variables and clauses required.

First we evaluate the expression on all 2^n possible inputs to generate a truth table. It is then processed using an external truth table minimization tool *Espresso* [Rudell and Sangiovanni-Vincentelli, 1987], which generates a list of clauses that can be used to encode the expression. When this expression is used anywhere in the model a node is created in the boolean

Optimizations		Time	Optimizations		Time	
χ step	θ step	Mean	χ step	θ step	t	p
Off	Off	7.5	Off	On	3.21	< 0.01
			On	Off	0.52	0.95
			On	On	2.11	0.15
Off	On	8.5	On	Off	3.63	< 0.01
			On	On	0.98	0.75
On	Off	7.4	On	On	2.56	0.05
On	On	8.2				

Figure 1: Pairwise comparison of four *SHA-3-512* optimization combinations using the Games-Howell procedure. Only measurements for more than 12 rounds were used to avoid randomness in timing, for a total of $n = 949$ samples per strategy.

circuit representation. It behaves like any other node and can be used in further expressions, however during instance generation the optimized list of clauses is used instead of the naïve Tseitin encoding.

3.1.1 SHA-1 analysis

We evaluated the effectiveness of this optimization on preimage attacks on *SHA-1* using the unmodified Tseitin encoding and using the *Espresso* optimizer on both the *choice* and *majority* round functions. We measured the running time of finding an 8-bit preimage for 32-bit input message. The preimage bits were obtained by hashing random messages to ensure a solution would exist even for instances with reduced number of rounds. We repeated the experiment multiple times for a total of 5670 samples for each variant.

Figure 2 shows the mean running times for both instances without optimizations and for ones using *Espresso* minimization. As we can see the effect of this optimization is quite small and the running times appear to be identical.

Using the *Games-Howell* post hoc test [Games and Howell, 1976] on instances with more than 20 rounds (to reduce the effect of randomness when measuring very short time intervals) we do obtain a mean time improvement of $t = 1.6$ seconds however at a fairly high significance level of $p = 0.12$ which does not give us enough evidence to reject the hypothesis that this optimization leads to no improvement.

3.1.2 SHA-3 analysis

We performed a similar experiment for the *SHA-3* hash function, finding an 8-bit preimage on 32-bit message where the preimage bits again came from randomly generated messages. We considered two possible optimizations in the round function and tested four variants – all combinations of turning on or off these two optimizations. A total of 2000 samples was collected for each variant.

The first optimization was minimizing the expression $x \oplus (\overline{y} \wedge z)$ in the χ step, which is used to fill the 5×5 state matrix S in each round.

The second optimization was in the θ step, where the C vector is filled using an exclusive or of five different values. Without optimization this leads to four extra variables and 16 extra clauses. Using the Espresso minimization will lead to 32 clauses but only one extra variable.

Using the Games-Howell procedure again we obtain the pairwise comparison shown in figure 1.

We can see that the *xor* optimization – which reduces the number of variables but requires twice as many clauses – in fact leads to higher solving time (mean difference $t = 3.21$) at significance level of $p < 0.01$. Thus the default behavior is more efficient. On the other hand, from the high p-value of 0.75 we can't reject the hypothesis that optimizing the χ step makes no difference at all.

3.1.3 Discussion

Even with large number of samples to eliminate the intrinsic randomness in SAT solving times we were unable to reject hypotheses that the tested optimizations do not lead to an improvement. From this we conclude that they do not provide significant benefits.

While our library provides this optimization feature to users as we saw in our measurements it is not necessary to use it. Therefore minimal changes

to existing, off-the-shelf implementations of hash functions (without the need to identify expressions with non-optimal Tseitin representation) are sufficient to match the hand-optimized instances such as in [Nossum, 2012].

3.2 Branching order

The most important heuristic in a SAT solver is the decision which unassigned variable to pick next. A bad order might assign randomly picked values to many variables before some conflict is found and the search tree depth will be quite high. This in turn leads to an increased solving time. On the other hand a good heuristic would pick variables in an order that causes many propagations and with an unsatisfiable assignment leads to conflicts quickly.

The branching order is picked by a heuristic in the SAT solver, which does not have additional knowledge about what these variables represent and how they relate to each other. By extending the input file format and the variable picking algorithm we can provide our own (partial) variable branching order.

3.2.1 Implementation

We added a new input line type to the *DIMACS CNF* file format. A line in the form

`b` v `0`

means that variable v should be branched on first. When multiple such lines are provided the order of branching is the same as the order of these lines in the input file. Using this we can provide a list of any number of variables in the order we want them to be picked for assignment.

We modified the popular *MiniSat* solver [Een and Sörensson, 2005] to be able to parse and store this list. Then we modified the selection of next branching candidate to first pick all these variables in the specified order. Once all have been assigned we fall back to the standard branching order algorithm.

We have also added support for this to our modeling library. Before the boolean circuit is transformed to a set of CNF clauses and written to a file, the user can specify arbitrary branching order using variables defined in the model of the cryptographic primitive.

3.2.2 SHA-3 analysis

For *SHA-3-512* we tested various branching orders on 8-bit reference preimage attack with number of rounds ranging from 0 to 24 (full SHA-3). For every number of rounds 10 different instances were generated and each was solved 10 times, for a total of 100 samples per round per strategy. We compared the following branching order strategies:

none: No branching order was specified. This is the default unmodified MiniSat behavior.

r0-S-x-y: The S matrix from the first round is branched on first, in column-major order.

r0-S-y-x: Same as previous one, but in row-major order.

rlast-S-x-y and ***rlast-S-y-x-***: Same as previous two, but the S matrix from the last round is used instead.

The figures 3 and 4 show *violin plots* of the distribution of ratios of the solving time and the number of conflicts. Each *violin* also show the mean, median and extreme values.

From these plots we see that while the ratios for the number of conflict are mostly below 1 (meaning that the *r0-S-x-y* strategy leads to fewer conflicts), the time ratios are often higher than 1.

The Games-Howell procedure (figure 5) confirms these findings, with the mean difference for number of conflicts between the *none* and *r0-S-x-y* strategies of $t = 17$ at significance level $p < 0.01$. However, for the solving time the high p-value does not let us reject the hypothesis that any difference is due to chance. Note that once again only samples for more than 12 rounds were included to avoid the strong effect of randomness for instances that solve in very short time.

The *r0-S-y-x* strategy behaves the same as *r0-S-x-y* – the number of conflicts for every instance is identical and the differences in time are negligible. On the other hand the strategies starting with the last round's S matrix lead to the same behavior as not providing any branch ordering at all (the *none* strategy).

Similar experiments were performed with the auxiliary vectors and matrices B, C and D with the same results – enforcing branching order did not lead to better solving times.

3.2.3 Discussion

The fact that these branching order do not change the solving time significantly must mean that either their choice does not lead to any forced assignments and

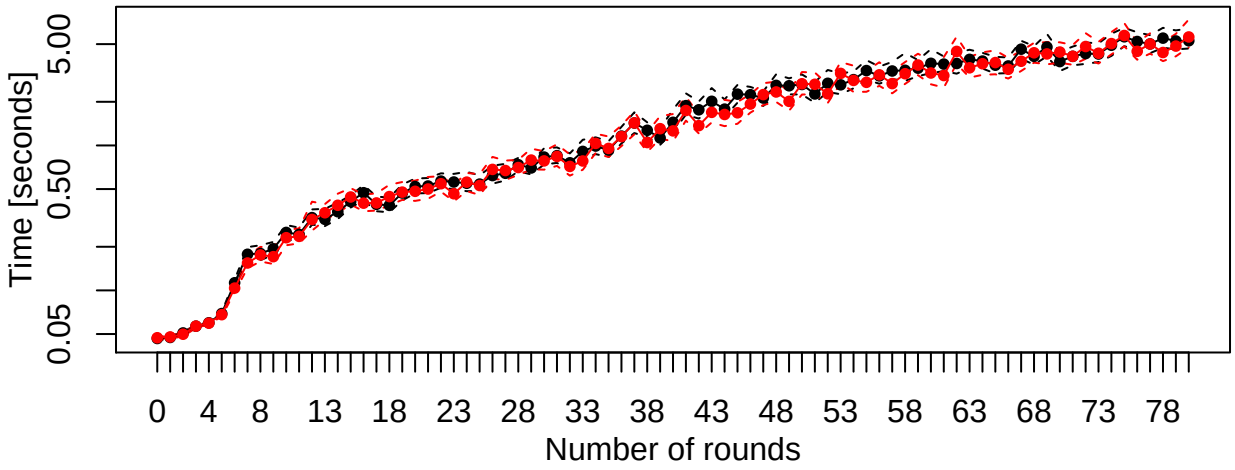

Figure 2: Mean running time and 95% confidence intervals for 8-bit preimage attack on *SHA-1* without optimizations (black) and using Espresso minimization for rounds functions (red).

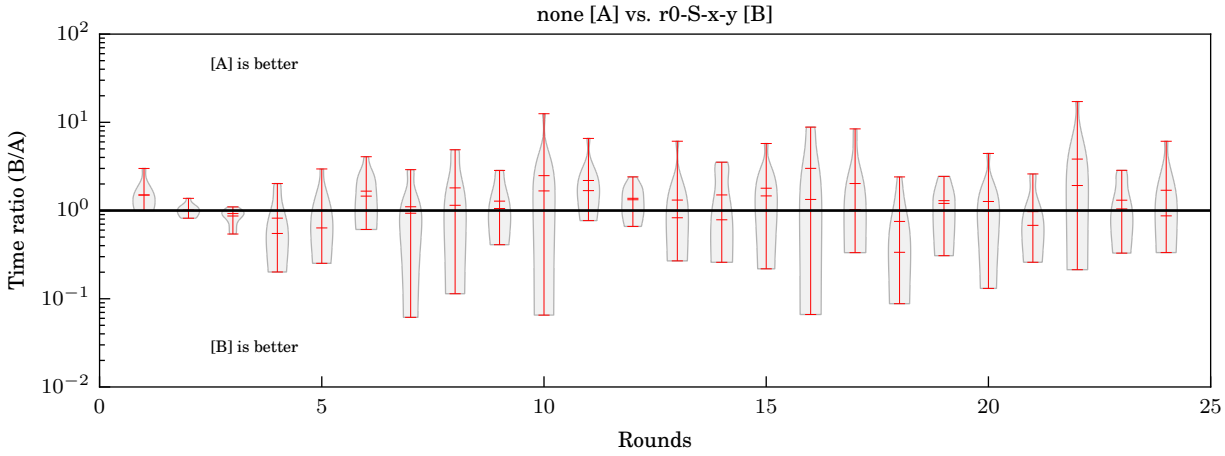

Figure 3: Violin plot showing the ratios distribution of solving time for the *none* and *r0-S-x-y* strategies.

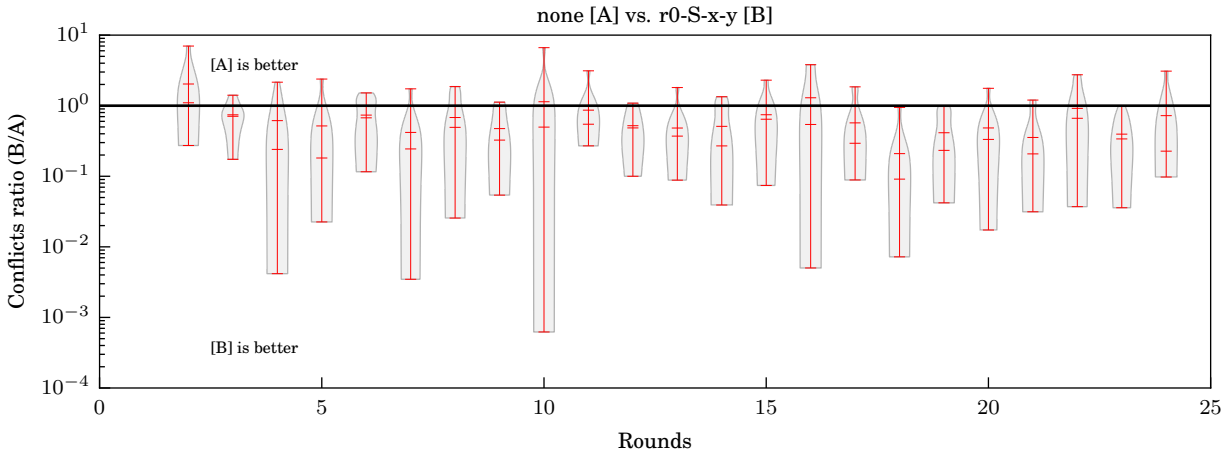

Figure 4: Violin plot showing the ratios distribution of number of conflicts for the *none* and *r0-S-x-y* strategies.

Strategy	Time	Conflicts	Strategy	Time		Conflicts	
	Mean			t	p	t	p
none	7.8	731	*r0-S-x-y*	0.65	0.97	17	< 0.01
			r0-S-y-x	0.58	0.98	17	< 0.01
			rlast-S-x-y	0.18	> 0.99	958	> 0.99
			rlast-S-y-x	0.03	> 0.99	958	> 0.99
r0-S-x-y	7.5	250	*r0-S-y-x*	0.07	> 0.99	0	< 0.01
			rlast-S-x-y	0.83	0.92	17	< 0.01
			rlast-S-y-x	0.68	0.96	17	< 0.01
r0-S-y-x	7.5	250	*rlast-S-x-y*	0.76	0.94	17	< 0.01
			rlast-y-x	0.60	0.97	17	< 0.01
r0-last-x-y	7.8	731	*rlast-S-y-x*	0.15	> 0.99	0	> 0.99
r0-last-y-x	7.8	731					

Figure 5: Pairwise comparison of various branching order strategies for *SHA-3-512* using the Games-Howell procedure. Only measurements for more than 12 rounds were used to avoid randomness in timing for a total of $n = 480$ samples per strategy.

conflicts (which is highly unlikely) or that the default MiniSat heuristic is also picking them for branching first. The second case means that this optimization is unnecessary and that the default SAT solver heuristic is sufficient in this case.

4 Conclusions

We have created a library for modeling various problems as SAT instances that can then be solved with SAT solvers. While the library is fairly universal we have specifically focused on cryptographic problems such as modeling preimage attacks on hash functions. To simplify this use case we make use of operator overloading that allows using our library with existing implementations with minimal changes.

Previous works in this area used handwritten code to generate instances of a specific hash function that were not easily modifiable. However this allowed them to optimize the resulting instances, reducing the number of variables and clauses required compared to a naïve Tseitin transformation. We added an option to optimize specific expression to our library and evaluated the effects of those optimizations.

While we found that the optimized instances can lead to fewer conflicts the solving time was not improved in any statistically significant way. This leads us to conclude that these optimizations are not required and therefore the minimal changes to existing implementations mentioned above are sufficient to create a reasonable instance.

We then modified the MiniSat solver to see if overriding the default branching order heuristic with the help of additional information about the problem structure would lead to speed improvements. However, same as with previous optimizations, we found that only the number of conflicts was reduced in this way. From this we conclude that the existing heuristics employed by modern solvers behave reasonably on these instances and therefore using an off-the-shelf solver is sufficient.

References

[Een and Sörensson, 2005] Een, N. and Sörensson, N. (2005). MiniSat: A SAT solver with conflict-clause minimization. *Sat*, 5.

[Games and Howell, 1976] Games, P. A. and Howell, J. F. (1976). Pairwise multiple comparison procedures with unequal ns and/or variances: a monte carlo study. *Journal of Educational and Behavioral Statistics*, 1(2):113–125.

[Jovanovic and Janicic, 2005] Jovanovic, D. and Janicic, P. (2005). Logical analysis of hash functions. In *Frontiers of Combining Systems, 5th International Workshop, FroCoS 2005, Vienna, Austria, September 19-21, 2005, Proceedings*, pages 200–215.

[Massacci and Marraro, 2000] Massacci, F. and Marraro, L. (2000). Logical cryptanalysis as a SAT problem. *J. Autom. Reasoning*, 24(1/2):165–203.

[Merkle et al., 1979] Merkle, R. C., Charles, R., et al. (1979). *Secrecy, Authentication, and Public Key Systems*. PhD thesis.

[Morawiecki and Srebrny, 2013] Morawiecki, P. and Srebrny, M. (2013). A sat-based preimage analysis of

reduced Keccak hash functions. *Inf. Process. Lett.*, 113(10-11):392–397.

[Nossum, 2012] Nossum, V. (2012). SAT-based preimage attacks on SHA-1.

[Pápay, 2016] Pápay, L. (2016). Modeling library, usage samples and analysis code. `http://dx.doi.org/10.5281/zenodo.50622`.

[Rudell and Sangiovanni-Vincentelli, 1987] Rudell, R. L. and Sangiovanni-Vincentelli, A. L. (1987). Multiple-valued minimization for PLA optimization. Technical Report 5.

[Tseitin, 1983] Tseitin, G. S. (1983). *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*, chapter On the Complexity of Derivation in Propositional Calculus, pages 466–483. Springer Berlin Heidelberg.