Abstract

SAT solvers are a universal tool for finding solutions to boolean satisfiability problems. In the past they have also 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 problems related to cryptographic hash functions, 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.

Abstrakt

SAT solvery sú univerzálny nástroj pre hľadanie riešení boolovských problémov splniteľnosti. V minulosti boli používané aj na kryptografické problémy, ako napríklad hľadanie vzoru hašovacích funkcií alebo kľúča prúdových šifier. Avšak tieto prístupy a riešenia nie je možné ľahko upraviť a opätovne využiť.

V našej práci sme vytvorili knižnicu pre jednoduché modelovanie SAT inštancií. Sústredili sme sa špeciálne na rôzne problémy súvisiace s kryptografickými hašovacími funkciami, ale vytvorená knižnica je dostatočne všeobecná a je ju možné použiť aj na iné účely.

S pomocou tejto knižnice sme vytvorili modely pre niekoľko kryptografických hašovacích funkcií. Ďalej sme testovali niekoľko SAT solverov, optimalizácií a heuristík a porovnávali sme ich efektivitu. Okrem iného sme využívali logický minimalizér *Espresso* na redukciu veľkosti inštancií, a tiež vlastné poradie vetvenia pri ohodnocovaní premenných s pomocou upraveného SAT solveru.

CHAPTER 1

Introduction

1.1 SAT solvers

SAT solvers (shortened from *satisfiability*) are programs which take a boolean satisfiability problem and find a 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 (a literal is either a variable or a negation of a variable) and the instance is a conjunction of such clauses. Thus all clauses must be true, and in each clause at least one literal must be true in order for the instance to be satisfied. 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 Implementation

Since SAT is an NP-complete language and assuming $P \neq NP$ we can not in general find solutions in deterministic polynomial time. A simple exponential time algorithm is to try every possible boolean truth assignment. To remember which assignments have been tried and which have not we can implement this algorithm as a recursive backtracking, where for every input variable we try in turn both possible values. This will only require linear memory to implement.

While there are no known algorithms that perform asymptotically better in he worst case, we can still improve on this simple exponential time algorithm significantly by avoiding paths that are guaranteed to lead to a *conflict* – a clause in which all literals have been assigned to false.

DPLL algorithm

First such improvement is the *DPLL algorithm* (Davis-Putnam-Logemann-Loveland, [DLL62; DP60]) which improves upon the naive backtracking algorithm by first applying two rules at each step of the recursion:

Unit propagation

We will call a clause a *unit* if it only contains one unassigned literal. If the clause is already satisfied we can ignore it during further search. Otherwise it can only be satisfied by assigning the necessary value to the remaining literal. This means there will be no branching for this literal and thus the search time will be reduced by avoiding the obviously wrong choice.

Pure literal elimination

We will call a literal *pure* if its variable only occurs in the formula with a single polarity (that is, always negated or always non-negated). All pure literals can be assigned the required value to make all clauses containing them true and these clauses can then also be ignored during further search.

After these rules are applied we continue as before by picking an unassigned variable and trying both possible truth value assignments recursively. These two simple rules will help to reduce the search space the algorithm has to look through and thus reduce the total running time.

Conflict-Driven Clause Learning

It can happen that after some partial assignment of variables there is already a conflict and it is not possible to extend this assignment to a satisfying one. However, the DPLL algorithm will continue to explore the entire search subspace. With *conflict-driven clause learning* (CDCL, [BS97; MS99]) we can avoid this by the use of *implication graph* and *learnt clauses*.

The *implication graph* is a directed graph in which vertices are variables with their assignment. These will be marked as either *decision* or *forced*.

At first we start with an empty implication graph and execute the DPLL algorithm. Every time we make an arbitrary decision (the recursive step in the algorithm) we will add a decision vertex representing this variable and the assignment we have chosen into the graph. Every time either of the two rules of the DPLL algorithm forces some

assignment of a variable we will add a forced vertex into the graph, with edges from every vertex that is a part of this forced choice.

For example, with clause $(x \vee y \vee z)$ and with partial assignment $x = 0, y = 0$ the unit propagation rule will force the assignment of $z = 1$. We will add this as a forced vertex, with edges from both $x = 0$ and $y = 0$.

When we add an vertex to the graph for some variable but this variable is already present with the opposite assignment we have found a conflict. Now comes the part of CDCL which speeds up this search – we will find all the decision vertices (variables) that are responsible for this conflict. Let us call them $x_1, \ldots, x_k$ with assignments $v_1, \ldots, v_k$. From this we can build a *learnt clause* L:

$$\begin{aligned}(x_1 = v_1 \wedge \cdots \wedge x_k = v_k) &\Rightarrow \bot \\ \overline{\bot} &\Rightarrow \overline{(x_1 = v_1 \wedge \cdots \wedge x_k = v_k)} \\ \top &\Rightarrow (x_1 = \overline{v_1} \vee \cdots \vee x_k = \overline{v_k}) \\ L &:= (x_1^{(\overline{v_1})} \vee \cdots \vee x_k^{(\overline{v_k})})\end{aligned}$$

The notation $x^{(v)}$ is used to represent x if $v = 1$ and $\overline{x}$ if $v = 0$. Symbols $\top$ and $\bot$ represent tautology and contradiction, respectively.

We can now add this learnt clause L into the list of clauses we have to satisfy, since not satisfying it is guaranteed to lead to a conflict. In the next step of the backtracking algorithm we will not return only a single level up from the recursion (that is, removing the assigning of just a single variable) but all the way back until we reach the first point where one of the conflicting variables was assigned a value.

This method is used in all modern SAT solvers and improves their performance significantly over the DPLL algorithm.

1.1.2 Interfacing with SAT solvers

Since our work depends on providing a suitable input to SAT solvers we will briefly describe the widely supported *DIMACS CNF* format[1] which we use. Later in section 3.3.1 we will describe augmenting this format for optimization purposes.

The file begins with a line in form

`p cnf` *vars 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 v_2 ... v_i `0`

[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.*

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 of a SAT solver 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.

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 stream 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.

More precisely, 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. For a more formal discussion of hash functions (which is beyond the scope of our work) we refer the reader to [RS04].

The properties which we require these functions to have are:

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

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*. Starting from the initial state (usually some constant) a part of the input message (with some form of padding) is processed and a new state produced. This is repeated until the whole input has been processed and the final state is then used to produce the digest.

Reduced hash functions and partial attacks

The expected time to find a preimage for actual hash functions as used in practical cryptography is usually so large we couldn't obtain a solution in a reasonable time. For this reason we will be working with *reduced* instances of hash functions – modifications that are similar but easier to attack. To create such reduced instances we can simply lower the number of rounds used in the computation.

Similarly, finding a full preimage where all the output digest bits are fixed to a certain value could take a long time. Instead we will focus on *partial* preimage attacks, where only a certain number of output digest bits are fixed.

The SHA hash functions

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* [Nat95] is a widely used hash function based on the *Merkle-Damgård* construction [M+79] with output digest size of 160 bits. The internal state consists of five 32-bit words A, B, C, D and E which are updated in 80 rounds.

After padding the message to a length that is multiple of 512 bits, it is processed in *chunks* of that same size. Each chunk is broken into 16 32-bit words $W_0, \ldots, W_{15}$. From those 64 additional words $W_{16}, \ldots, W_{79}$ are created by *extending* the chunk:

$$W_i = (W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16}) \lll 1 \quad (16 \leq i < 80)$$

where $\lll$ means cyclic left shift on 32-bit words.

These words are then processed in 80 rounds, during which the internal state is updated. In each round i, first the values f and k are computed based on the round number. Then T is computed as

$$T = (A \lll 5) + f + E + k + W_i$$

where addition is performed modulo 2^{32}.

Finally, the state is updated as follows

$$E \leftarrow D \qquad D \leftarrow C \qquad C \leftarrow B \lll 30 \qquad B \leftarrow A \qquad A \leftarrow T$$

The value of k is one of four constants, depending on the value of $\lfloor \frac{i}{20} \rfloor$ (that is, one value for rounds 0 to 19, another for rounds 20 to 39 and so on). Similarly, f is computed by one of four *round functions*:

$$\begin{aligned} f_0 &= Ch(B,C,D) = (B \wedge C) \oplus (\overline{B} \wedge D) \\ f_1 = f_3 &= B \oplus C \oplus D \\ f_2 &= Maj(B,C,D) = (B \wedge C) \oplus (B \wedge D) \oplus (C \wedge D) \\ f &= f_{\lfloor \frac{i}{20} \rfloor} \end{aligned}$$

After processing all chunks the resulting state is the output digest.

SHA-3 is a recently standardized [Dwo] hash function family based on the *Keccak* family [Ber+11] 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.

We will describe the steps θ and χ in little more detail since we will be referring to them later in section 4.2.2. In the θ step an auxiliary array C is filled as follows:

$$C_i = S_{i,0} \oplus S_{i,1} \oplus S_{i,2} \oplus S_{i,3} \oplus S_{i,4} \quad (0 \leq i < 5)$$

This is then used, along with two additional arrays B and D to compute the new values of the state S in the χ step:

$$A_{i,j} = B_{i,j} \oplus (\overline{B_{i+1,j}} \wedge B_{i+2,j}) \quad (0 \leq i, j < 5)$$

The most important difference compared to SHA-1 is that not the whole state is used for output. Instead it is divided into two parts: the 576-bit *bitrate* and 1024-bit *capacity*, and only the bitrate part is used for the output while the capacity part stays hidden.

The sponge construction then works in two phases – *absorbing* and *squeezing*. During absorbing, chunks of the padded input message are xored with the state and the 24 rounds are performed until the message is fully processed. Afterwards the bitrate part of the state is used for the output digest. If it is not long enough another series of

24 rounds is performed on the whole state and the resulting bitrate part is appended to the digest and so on until the desired length is reached.

This flexible design allows arbitrary output digest sizes for specific applications. It also makes reversing the process harder, since the entire output state is not known.

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 [Páp16] 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 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 $(x_1 \wedge \cdots \wedge x_k)$.

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* [Tse83]. 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 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 [MM00]. 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 [JJ05] 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] [MS13]. 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 not publicly available.

An in-depth analysis of *SHA-1* preimage attacks was done in [Nos12]. 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.

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

CHAPTER 2

Modeling library

The works mentioned in previous chapter 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 chapter 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 1.1.2) 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.

2.3 Using the library

We will demonstrate the use of our library on two examples. The first one is very simple and shows the entire process from start to finish. The second example shows a real-world use case – modifying existing *SHA-1* implementation to use our library for

instance generation. We compare this modified implementation to a hand-crafted one from a previous work.

2.3.1 Simple example

Suppose we wish to find solutions to the boolean equation $X = A \wedge (B \oplus C)$. We begin with importing the modeling library and defining the input variables A,B and C as 1-bit vectors:

```
1 from instance import *
2 A, B, C = BitVector(1), BitVector(1), BitVector(1)
```

Next, we can write the expression in the same way as we normally would. The resulting variable X will also be of type *BitVector* and will store the boolean circuit representing this expression:

```
3 X = A & (B ^ C)
```

Now we can create a new instance, generate it using the variables we are interested in and solve it using any SAT solver using the *DIMACS* standard. Afterwards the satisfying assignment is easily accessible through the variables. In this case we will simply print it out:

```
4 instance = Instance()
5 instance.emit([X])
6 instance.solve(['minisat'])
7
8 print([q.getValuation(instance) for q in [A, B, C, X]])
```

The output might look like `[[False], [False], [False], [False]]`, which is indeed a valid solution to our equation. However, suppose we wish the result X to be true. We can add this additional constraint by setting

```
4 X.bits = [True]
```

before generating and solving the instance. Now we obtain the output `[[True], [False], [True], [True]]` which is another valid solution with the additional property that the value of X is true.

The entire program is less than ten lines long and very straightforward. We will use the same two concepts – replacing input variables with *BitVector* and adding additional constraints – in the next example.

2.3.2 SHA-1 example

We begin with a standard *SHA-1* implementation based on one available online [Alt15]. We extended it to support reduced instances. Here we show the most important parts of this code:

```
1  # Round functions and constants
2  fs = [lambda a, b, c, d, e: (b & c) | (~b & d),
3        lambda a, b, c, d, e: b ^ c ^ d,
4        lambda a, b, c, d, e: (b & c) | (b & d) | (c & d),
5        lambda a, b, c, d, e: b ^ c ^ d]
6  K = [0x5A827999, 0x6ED9EBA1, 0x8F1BBCDC, 0xCA62C1D6]
7
8  def sha1(message, rounds = 80):
9      # Initialization
10     h0, h1, h2, h3, h4 = 0x67452301, ...,  0xC3D2E1F0
11     # (omitted) Message padding
12     for pos in range(0, len(message), 64):
13         # (omitted) Prepare message chunk W
14         A, B, C, D, E = h0, h1, h2, h3, h4
15         for i in range(rounds):
16             F = fs[i//20](A, B, C, D, E)
17             k = K[i//20]
18
19             T = (leftrotate(A, 5) + F + E + k + W[i]) & 0xFFFFFFFF
20             A, B, C, D, E = T, A, leftrotate(B, 30), C, D
21         h0 = (h0 + A) & 0xFFFFFFFF
22         # ...
23         h4 = (h4 + E) & 0xFFFFFFFF
```

To modify this code to use our library and produce a SAT instance we need to perform only a few small changes. First of all, we need to convert all constants to *BitVector* objects of the appropriate size:

```
6  K = [intToVector(x) for x in [0x5A827999, ...]]
10 h0, h1, h2, h3, h4 = [intToVector(x) for x in [0x67452301, ...]]
```

Next we replace the `leftrotate` function with the *CyclicLeftShift* provided by our library[1]. Since addition on 32-bit *BitVector* objects is automatically done modulo 2^{32} we can remove the unnecessary *and mask*:

```
15     for i in range(rounds):
16         F = fs[i//20](A, B, C, D, E)
17         k = K[i//20]
18
19         T = CyclicLeftShift(A, 5) + F + E + k + Mvec[i]
20         A, B, C, D, E = T, A, CyclicLeftShift(B, 30), C, D
21     h0, h1, h2, h3, h4 = h0+A, h1+B, h2+C, h3+D, h4+E
```

As we can see the changes required are minimal and trivial to perform. For brevity we have omitted details such as message padding – the full source code can be found in the attachment [Páp16]. It is about 80 lines long and includes additional constraints on the output bits to find (partial) preimages. After obtaining a solution the message is extracted and hashed using a reference implementation to ensure its validity.

2.3.3 Comparison to existing work

In [Nos12] a custom program of about 800 lines is used to generate instances for preimage attacks on *SHA-1*. Our implementation using the modeling library we created is about ten times shorter. It also includes verification of the solution, unlike in Nossum's work where two additional programs are required to parse and verify the SAT solver output.

In addition to requiring much less code to be written our approach is also much more readable. Compare the code for the innermost loop where one of the four *SHA-1* round functions is computed, shown in figure 2.1.

For clarity and fairness of comparison we modified our version of the code slightly to avoid *Python* specific features such as *lambda functions* which make our code even shorter. It is clear that our approach leads to much more readable code and is easier to write.

[1] Since Python does not provide an operator for cyclic left shift we can't use operator overloading in this case.

(a) Instance generating tool from [Nos12]

```
if (i >= 0 && i < 20) {
  for (unsigned int j = 0; j < 32; ++j) {
    clause(-f[j], -b[j], c[j]);
    clause(-f[j], b[j], d[j]);
    clause(-f[j], c[j], d[j]);

    clause(f[j], -b[j], -c[j]);
    clause(f[j], b[j], -d[j]);
    clause(f[j], -c[j], -d[j]);
  }
} else if (i >= 20 && i < 40) {
  xor3(f, b, c, d);
} else if (i >= 40 && i < 60) {
  for (unsigned int j = 0; j < 32; ++j) {
    clause(-f[j], b[j], c[j]);
    clause(-f[j], b[j], d[j]);
    clause(-f[j], c[j], d[j]);

    clause(f[j], -b[j], -c[j]);
    clause(f[j], -b[j], -d[j]);
    clause(f[j], -c[j], -d[j]);
  }
} else if (i >= 60 && i < 80) {
  xor3(f, b, c, d);
}
```

(b) Adapted version of our instance generating code

```
if 0 <= i < 20:
  f = (b & c) | (~b & d)
elif 20 <= i < 40:
  f = b ^ c ^ d
elif 40 <= i < 60:
  f = (b & c) | (b & d) | (c & d)
else:
  f = b ^ c ^ d
```

Figure 2.1: Comparison of code for computing the *SHA-1* round functions.

For example, for the *choice* round function used in first 20 rounds we simply use the definition provided in the standard (see section 1.2.1). On the other hand, a set of six clauses to encode this function had to be found and used in the referenced work.

As we will discuss in section 3.2, our approach does lead to a less efficient encoding (in terms of number of variable and clauses in the resulting instance). However, as experiments in 4.2 demonstrate, this does not have any measurable effect on the solving time. Moreover, we added expression optimization support to our library that can be used to reduce the instances and in the case of the *choice* function does in fact lead to an even more efficient encoding using just four clauses.

2.3.4 The N-Queens problem

As we have stated previously, even though we have so far focused on hash functions, our library is generic enough to be used for other problems. We demonstrate this with the following simple example – solving the famous *N-Queens* problem.

In this problem we have a chessboard of size $N \times N$ and the task is to place upon it N queens such that no two of them threaten each other.

First of all we define the parameter N and declare the board, which we will represent as N rows, each one being an N-bit vector. *True* bits will mean a queen is placed there, *false* bits will represent empty squares.

```
7 N = 8
8 board = [BitVector(N) for _ in range(N)]
```

Now we need to enforce the necessary constraints. We begin with the rule that there must be at least one queen in every row (which follows easily from the problem statement). We can model this in the following way: we take the binary *or* of every cyclic rotation of the row. If there is at least one true bit anywhere in the row, all bits of the result will be true, otherwise all will be false. We can then force all the bits of the result to be true.

```
12 for row in board:
13     rot = FalseVector(N)
14     for i in range(N):
15         rot |= CyclicLeftShift(row, i)
16     rot.bits = [True]*N
```

Next we need to make sure there is at most one true bit in each row, which we do in a similar way. Suppose there are two true bits in the row separated by k false bits.

If we take the binary *and* of the row and the cyclic left shift of the row by k bits we will obtain a vector with one true bit. Since we do not know the value of k we will take the binary *or* of all such vectors for all values of $0 < k < N$:

```
20 for row in board:
21     rot = FalseVector(N)
22     for i in range(1, N):
23         rot |= row & CyclicLeftShift(row, i)
24     rot.bits = [False]*N
```

We will do a similar trick to make sure there is at most one queen on every diagonal. We omit the code here for brevity.

Lastly we need to make sure there is precisely one queen in every column. However since we already have rules to enforce precisely one queen in every row, it is enough to ensure *at least one* queen in every column:

```
38 row_or = FalseVector(N)
39 for row in board:
40     row_or |= row
41 row_or.bits = [True]*N
```

Full source code for this sample is available in the attachment (see appendix A) and is just 47 lines long including comments. The output of the program shows the solution as a board, with the symbol # representing a queen:

```
$ python3 test_nqueens.py
========================[ Problem Statistics ]=========================
|  Number of variables:          5384                                  |
|  Number of clauses:           12104                                  |
=======================================================================
CPU time              : 0.008 s

SATISFIABLE
...#....
.#......
......#.
..#.....
.....#..
.......#
```

```
....#...
#.......
```

Even large instances, with values of N around 50, can be solved in just a few seconds using this program.

2.4 Hash function toolkit

In addition to the modeling library we also provide a simple command line interface for generating custom instances called *hashtoolkit*. It allows us to specify the hash function, number of rounds and input message length. Additionally the output digest bits can individually be set to either true, false or r – the same as a randomly generated reference digest.

In this example we find a 10-round 64-bit preimage on *SHA-1*, where the first eight bits are equal to a reference message and next eight bits are equal to the hexadecimal byte $0f$:

```
$ python3 hashtoolkit.py -h sha1 -l 64 -r 10 \
  -o 'rrrrrrrr00001111' -s 'minisat'
...
SATISFIABLE
Reference message: b'\x06)\xeaVqz\xe1\x8a'
Reference digest: 44c79c8b97df9297a4f551b3d14ec9aafe296a32

Message length 64 bits
Message bytes: b'NQ\xdbd\x8a\x06\x98\xde' rounds:  10
Message digest:   440f7115cc0483f042885b32247fe5eab998a8b4
```

Collisions can also be found using this tool. Supported hash functions are *MD5*, *SHA-1* and *SHA-3-512*. More details about the tool and usage instructions can be found in appendix B.

CHAPTER 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 heuristics which lead to decrease of solving time.

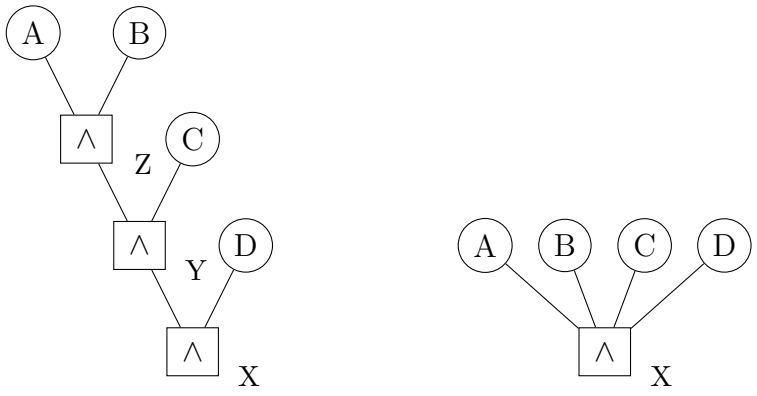
In this chapter we will describe several optimizations that we implemented and evaluated. Some of these involve a simple change in output instance generation, others involve changes to the SAT solvers themselves.

3.1 Merging operators

Since all operators in the *Python* programming language are either unary or binary so are all the nodes in the created boolean circuit. Writing an expressions such as `X = A & B & C & D` will result in the boolean circuit shown in figure 3.1(a) since the *and* operator is left-to-right associative.

A simple recursive algorithm can be used to walk over the boolean circuit in reverse topological order and merge such structures into a single n-ary node. At each step, if one or both of the operands of some boolean operator node are nodes of the same type, they can be merged together to a single node. After repeating this procedure for the whole tree we obtain the result shown in figure 3.1(b).

We implemented such optimization in our library and modified the clause generation code to support arbitrary arities. This is a simple optimization that can reduce the number of variables and clauses required to encode the given model as a SAT instance. Next we describe a more advanced method that can optimize not just sequences of identical operators but arbitrary expressions.



(a) Before merging (b) After merging

Figure 3.1: Comparing the effect of operator merging on the boolean circuit resulting from the expression `X = A & B & C & D`.

3.2 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:

$$\begin{aligned}
(\overline{a} \vee x) \wedge (\overline{a} \vee y) \wedge (a \vee \overline{x} \vee \overline{y}) &\qquad \text{Left AND gate} \\
(\overline{b} \vee \overline{x}) \wedge (\overline{b} \vee z) \wedge (b \vee x \vee \overline{z}) &\qquad \text{Right AND gate} \\
(c \vee a \vee \overline{b}) \wedge (c \vee \overline{a} \vee b) \wedge (\overline{c} \vee a \vee b) \wedge (\overline{c} \vee \overline{a} \vee \overline{b}) &\qquad \text{XOR gate}
\end{aligned}$$

A better encoding using just six clauses and one variable is shown in [Nos12]:

$$\begin{aligned}
&(\overline{a} \vee \overline{x} \vee y) \wedge (\overline{a} \vee x \vee z) \wedge (\overline{a} \vee y \vee z) \\
\wedge\ &(a \vee \overline{x} \vee \overline{y}) \wedge (a \vee x \vee \overline{z}) \wedge (a \vee \overline{y} \vee \overline{z})
\end{aligned}$$

In fact, it is possible to encode this expression with just four clauses and one variable:

$$(a \vee \overline{x} \vee \overline{y}) \wedge (a \vee x \vee \overline{z}) \wedge (\overline{a} \vee x \vee z) \wedge (\overline{a} \vee \overline{x} \vee y)$$

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.

```
# Unoptimized expression
f = (b & c) | (~b & d)

# Optimized using Espresso
choice = OptimizeExpression(lambda b, c, d: (b & c) | (~b & d))
f = choice(b, c, d)
```

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* [Rud86], 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 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.

Using this optimization on the *SHA-1 choice* round function does indeed lead to the minimal encoding with just four clauses as shown above. Similarly, some steps of the *SHA-3* hash function can be wrapped and optimized in this way. Both of these optimizations are further described and evaluated in section 4.2.

3.3 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.

3.3.1 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 [ES05] 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 also made similar changes to the *CryptoMiniSat* solver [SNC09], which is based on MiniSat.

We have also added support for this feature 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.

CHAPTER 4

Experiments

In this chapter we describe various experiments that were performed to evaluate the effectiveness and necessity of optimizations described in the previous chapter. These experiments compare the time required to solve particular instances across two or more variants. Each variant can be either a modification to the instance itself (by changing the way it is generated), or the instance can be unchanged but rather the behavior of the SAT solver itself is modified.

4.1 Methodology

Since the running time of a SAT solver is not deterministic it is not enough to simply perform a single run for every instance. In fact, given the same instance and the same SAT solver the running time can differ significantly, even by several orders of magnitude. Multiple samples therefore have to be gathered and sound statistical methods used to discover patterns in this noisy data.

Sampling procedure

To counter the high variability in time even for one instance we will solve each instance several times.

Additionally, not all instances for particular parameters (such as number of rounds) are equally hard. For example, reduced hash functions with significantly lower number of rounds than the full hash function do not behave sufficiently as random functions. Some output digests might be significantly less likely to be produced than other, or might not be even be possible. When we are performing a partial preimage attack by fixing several digest bits to be equal to a reference digest (obtained by first hashing a random message) some instances might have fewer solutions than others and be therefore harder.

For this reason we also perform all such experiments on multiple instances that have identical parameters but the random reference messages are independently generated.

Censoring

Sometimes we might obtain an instance that takes very long time to solve. To ensure that experiments finish in a reasonable time it is common to enforce a time limit and abort all computations that exceed it. Since in such cases we can not know the true running time we must discard these aborted measurements.

Data sets with such discarded measurements are called *censored*. They make statistical analysis complicated if not impossible, since even trivial statistics such as mean can not be computed without knowing the true censored values.

For this reason we avoided censoring in our experiments. While we did have a time limit in place, all experiments that contained censored runs were discarded and not considered for analysis. To obtain valid experiments we either increased the time limit or reduced some of the parameters (such as number of rounds or length of the preimage) until we got a configuration that finished without any censoring.

Statistical analysis

For plots of solving time versus some parameter of the instance the most useful statistic is the mean. It shows the general trend in behavior and the rate with which the time increases. However it is by itself unreliable since the random variability can affect it significantly.

For this reason we also include 95% confidence intervals in our plots. By performing more and more repetitions and obtaining more samples we can reduce the size of those intervals and obtain reliable results.

It is important to note that standard techniques for computing means and confidence intervals require either normal distribution of the data, or at least some knowledge about the distribution. With SAT solving times we have neither – although the solving times for some cryptographic instances have been shown [BCJ07a] to have a log-normal distribution we would still have to verify this assumption for our data sets.

Instead we use the BC_a bootstrap procedure [DE96] for calculating means and confidence intervals that does not require any knowledge or assumptions about the distribution.

To evaluate the effect of some optimization on the running time we need a statistical procedure to decide if there is a significant difference in the running time distribution or if all differences can be attributed to chance. Since experiments can have varying parameters (such as number of rounds) and for each parameter combination we run multiple independent samples a pairwise sample comparison is required. There are several such procedures, such as the Student's t-test or the Tukey's test.

However, both of those make assumptions about the underlying distributions which are not met in our case. For that reason we use the rather less known *Games-Howell* procedure [GH76] which does not require these assumptions.

Both used procedures are implemented by the *R* statistical computing language [R C13]. Availability of implementations was also a factor when deciding on methodology – using a tested implementation greatly reduces the chance of errors.

4.2 Expression encoding evaluation

In this section we discuss the effect of expression encoding optimizations described in section 3.2. We performed this experiment on two hash functions, the *SHA-1* and *SHA-3*.

4.2.1 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 4.2 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 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.

4.2.2 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.

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 4.1: 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.

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 4.1.

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.

4.2.3 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 [Nos12].

4.3 Branching order evaluation

For branching order optimization described in section 3.3 we used the *SHA-3* hash function for evaluation.

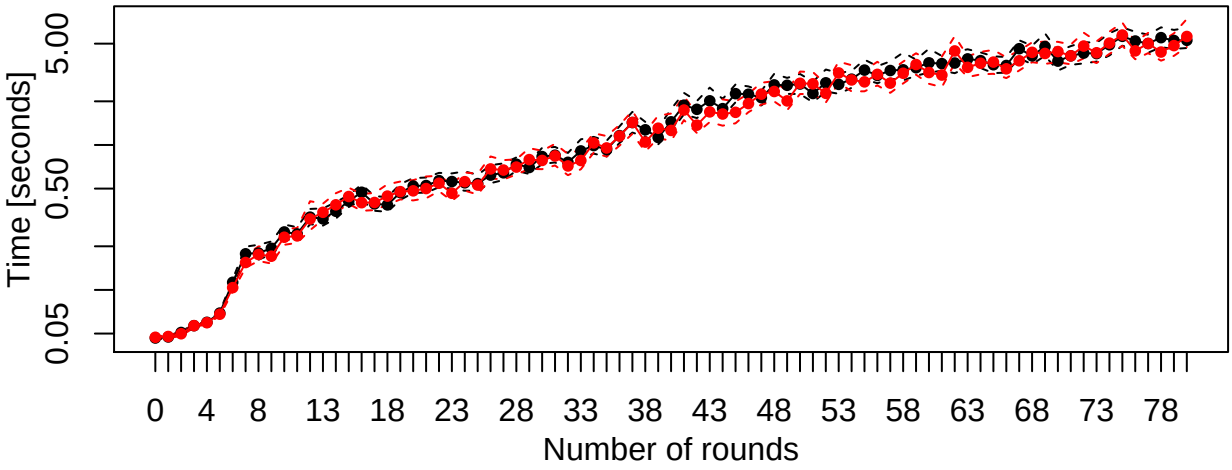


Figure 4.2: Mean running time and 95% confidence intervals for 8-bit preimage attack on *SHA-1* without optimizations (black) and using Espresso minimization for rounds functions (red).

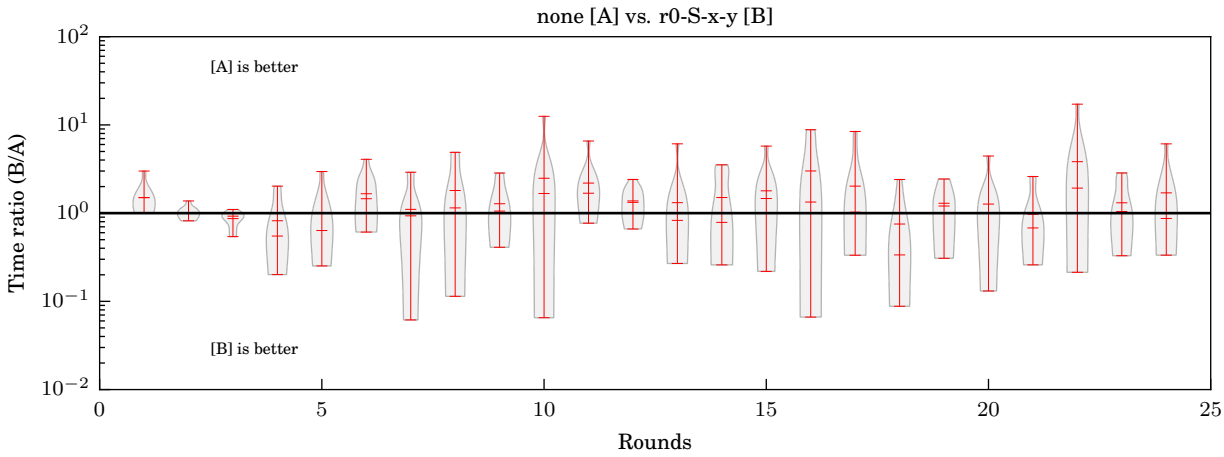


Figure 4.3: Violin plot showing the ratios distribution of solving time for the *none* and *r0-S-x-y* strategies.

4.3.1 SHA-3 analysis

For *SHA-3-512* we tested various branching orders on 8-bit reference preimage attack with number of rounds ranging from 1 to 24 (the maximum for this hash function). 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.

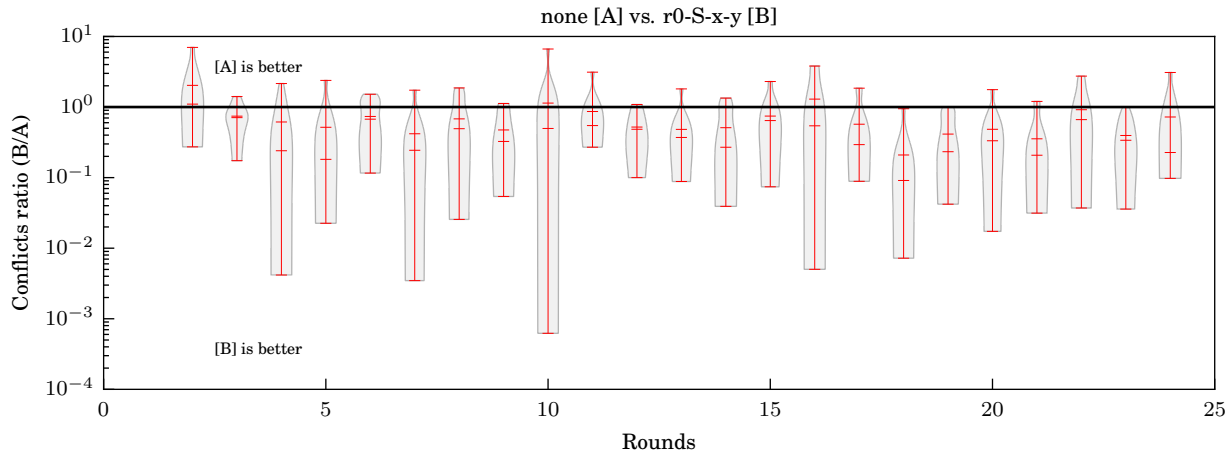


Figure 4.4: Violin plot showing the ratios distribution of number of conflicts for the *none* and *r0-S-x-y* strategies.

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 4.5: 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.

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 4.3 and 4.4 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 4.5) 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.

4.3.2 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 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.

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.

APPENDIX A

Library

The library we developed is available at

`https://github.com/lacop/master-thesis-code`

or on the CD attachment [Páp16].

The attachment includes:

Modeling library *instance.py*
: The most important part of the library, used to model boolean circuits, solve them using SAT solver and read the solutions.

Optimizations *optimizations.py*
: Implementation of the operator merging (section 3.1) and the *Espresso* expression minimization (section 3.2) optimizations.

HashToolkit *hashtoolkit.py, hashes.py*
: The *HashToolkit* command line tool (appendix B).

Samples *samples/*
: Sample implementations of several hash functions.

Statistics and plots *plots/*
: Code used for statistical analysis and to generate plots in this text.

APPENDIX B

HashToolkit

The *HashToolkit* command line tool can be used to generate, solve and verify (partial) preimage or collision attacks on hash functions. Supported hash functions are *MD5*, *SHA-1* and *SHA-3-512*.

Input and output bits can be individually set to arbitrary values. Output bits can additionally be set to be the same as a randomly generated reference digest.

Moreover, the length of the input message and number of rounds can also be customized. Any SAT solver that supports the *DIMACS* format (section 1.1.2) can be used for solving.

For more details about all the commands line options run the tool with the `--help` flag.

Bibliography

[Alt15] AJ Alt: *An Implementation of the SHA-1 Hashing Algorithm in Pure Python*, `https://github.com/ajalt/python-sha1`, 2015.

[BCJ07a] Gregory V Bard, Nicolas T Courtois, Chris Jefferson: *Efficient methods for conversion and solution of sparse systems of low-degree multivariate polynomials over GF (2) via SAT-solvers*, in: (2007).

[BCJ07b] Gregory V Bard, Nicolas T Courtois, Chris Jefferson: *Efficient methods for conversion and solution of sparse systems of low-degree multivariate polynomials over GF (2) via SAT-solvers*, in: (2007).

[Ber+11] Guido Bertoni et al.: *The keccak reference*, in: *Submission to NIST (Round 3)* 13 (2011).

[BHM09] Armin Biere, Marijn Heule, Hans van Maaren: *Handbook of satisfiability*, vol. 185, IOS press, 2009.

[BS97] Roberto J Bayardo Jr, Robert Schrag: *Using CSP look-back techniques to solve real-world SAT instances*, in: *AAAI/IAAI*, 1997, pp. 203–208.

[Cla+08] Koen Claessen et al.: *SAT-solving in practice*, in: *Discrete Event Systems, 2008. WODES 2008. 9th International Workshop on*, IEEE, 2008, pp. 61–67.

[DE96] Thomas J DiCiccio, Bradley Efron: *Bootstrap confidence intervals*, in: *Statistical science* (1996), pp. 189–212.

[DLL62] Martin Davis, George Logemann, Donald Loveland: *A machine program for theorem-proving*, in: *Communications of the ACM* 5.7 (1962), pp. 394–397.

[DP60] Martin Davis, Hilary Putnam: *A computing procedure for quantification theory*, in: *Journal of the ACM (JACM)* 7.3 (1960), pp. 201–215.

[Dwo] Morris J Dworkin: *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*, in: *Federal Inf. Process. Stds.(NIST FIPS)-202 (August 2015)* ().

[EPV08] Tobias Eibach, Enrico Pilz, Gunnar Völkel: *Attacking Bivium using SAT solvers*, in: *Theory and Applications of Satisfiability Testing–SAT 2008*, Springer, 2008, pp. 63–76.

[ES05] Niklas Een, Niklas Sörensson: *MiniSat: A SAT solver with conflict-clause minimization*, in: *Sat* 5 (2005).

[FKD09] RT Faizullin, IG Khnykin, VI Dylkeyt: *The SAT solving method as applied to cryptographic analysis of asymmetric ciphers*, in: *arXiv preprint arXiv:0907.1755* (2009).

[FSK12] Niels Ferguson, Bruce Schneier, Tadayoshi Kohno: *Cryptography engineering: design principles and practical applications*, John Wiley & Sons, 2012.

[GH76] Paul A Games, John F Howell: *Pairwise multiple comparison procedures with unequal n's and/or variances: a Monte Carlo study*, in: *Journal of Educational and Behavioral Statistics* (1976).

[Hom+12] Ekawat Homsirikamol et al.: *Security margin evaluation of SHA-3 contest finalists through SAT-based attacks*, in: *Computer Information Systems and Industrial Management*, Springer, 2012, pp. 56–67.

[JJ05] Dejan Jovanović, Predrag Janičić: *Frontiers of Combining Systems: 5th International Workshop, FroCoS 2005, Vienna, Austria, September 19-21, 2005. Proceedings*, in: Springer Berlin Heidelberg, 2005, chap. Logical Analysis of Hash Functions, pp. 200–215, DOI: 10.1007/11559306_11, URL: http://dx.doi.org/10.1007/11559306_11.

[JK10] Philipp Jovanovic, Martin Kreuzer: *Algebraic attacks using SAT-solvers*, in: *Groups–Complexity–Cryptology* 2.2 (2010), pp. 247–259.

[KL07] Jonathan Katz, Yehuda Lindell: *Introduction to modern cryptography: principles and protocols*, CRC Press, 2007.

[LHS04] Bin Li, Michael S Hsiao, Shuo Sheng: *A novel SAT all-solutions solver for efficient preimage computation*, in: *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, vol. 1, IEEE, 2004, pp. 272–277.

[M+79] Ralph Charles Merkle, Ralph Charles, et al.: *Secrecy, authentication, and public key systems*, in: (1979).

[MM00] Fabio Massacci, Laura Marraro: *Logical cryptanalysis as a SAT problem*, in: *Journal of Automated Reasoning* 24.1-2 (2000), pp. 165–203.

[MS13] Paweł Morawiecki, Marian Srebrny: *A SAT-based preimage analysis of reduced KECCAK hash functions*, in: *Information Processing Letters* 113.10 (2013), pp. 392–397.

[MS99] João P Marques-Silva, Karem A Sakallah: *GRASP: A search algorithm for propositional satisfiability*, in: *Computers, IEEE Transactions on* 48.5 (1999), pp. 506–521.

[MVV10] Alfred J Menezes, Paul C Van Oorschot, Scott A Vanstone: *Handbook of applied cryptography*, CRC press, 2010.

[MZ06] Ilya Mironov, Lintao Zhang: *Applications of SAT solvers to cryptanalysis of hash functions*, in: *Theory and Applications of Satisfiability Testing-SAT 2006*, Springer, 2006, pp. 102–115.

[Nat95] National Institute of Standards and Technology: *FIPS PUB 180-1: Secure Hash Standard*, 1995, URL: `http://www.itl.nist.gov/fipspubs/fip180-1.htm`.

[Nos12] Vegard Nossum: *SAT-based preimage attacks on SHA-1*, in: (2012).

[Páp16] Ladislav Pápay: *Modeling library, usage samples and analysis code*, `http://dx.doi.org/10.5281/zenodo.48832`, Apr. 2016, DOI: `10.5281/zenodo.48832`.

[Pat13] Constantinos Patsakis: *RSA private key reconstruction from random bits using SAT solvers*, in: *IACR Cryptology ePrint Archive* 2013 (2013), p. 26, URL: `https://eprint.iacr.org/2013/026`.

[R C13] R Core Team: *R: A Language and Environment for Statistical Computing*, ISBN 3-900051-07-0, R Foundation for Statistical Computing, Vienna, Austria, 2013, URL: `http://www.R-project.org/`.

[RS04] Phillip Rogaway, Thomas Shrimpton: *Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance*, in: *Fast Software Encryption*, Springer, 2004, pp. 371–388.

[Rud86] Richard L Rudell: *Multiple-valued logic minimization for PLA synthesis*, tech. rep., DTIC Document, 1986.

[Sch07] Bruce Schneier: *Applied cryptography: protocols, algorithms, and source code in C*, john wiley & sons, 2007.

[Sen14] Gustav Sennton: *On conflict driven clause learning–a comparison between theory and practice*, in: (2014).

[SNC09] Mate Soos, Karsten Nohl, Claude Castelluccia: *Extending SAT solvers to cryptographic problems*, in: *Theory and Applications of Satisfiability Testing-SAT 2009*, Springer, 2009, pp. 244–257.

[Tse83] Grigori S Tseitin: *On the complexity of derivation in propositional calculus*, in: *Automation of Reasoning*, 1983.