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 [1]. 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 [33] in file *samples/sha1_instance_test.py*. It also 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 [32] 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 [32]

```
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 4.3 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) in file *samples/nqueens.py*. The output of the program shows the solution as a board, with the symbol # representing a queen:

```
$ python3 test_nqueens.py
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.

```
# 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* [37], 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 4.3.

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 [12] 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 [41], which is based on MiniSat. However since we got better performance from using MiniSat only this solver was used for further evaluation.

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:

```
1 instance.assignVars(all_variables)
2
3 for var in variables_for_branching:
4     instance.branch(var)
5
6 instance.emit(all_variables)
7 instance.solve(['minisat'])
```

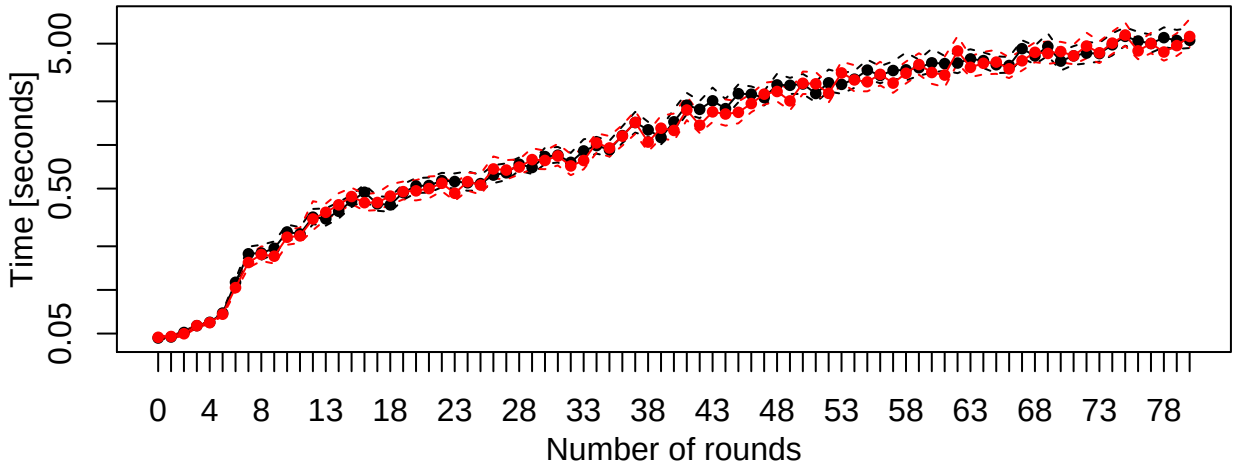

Figure 4.1: Mean running time and 95% confidence intervals for 8-bit preimage attack on *SHA-1* without optimizations (black) and using Espresso minimization for rounds functions (red).

4.3.1 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.

This reduces the number of clauses required for a full 80 round instance by about ten thousand from approximately 210 thousand in the unoptimized case. Similarly the number of variables is reduced by about four thousand from approximately 43 thousand.

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 4.1 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.

4.3.2 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

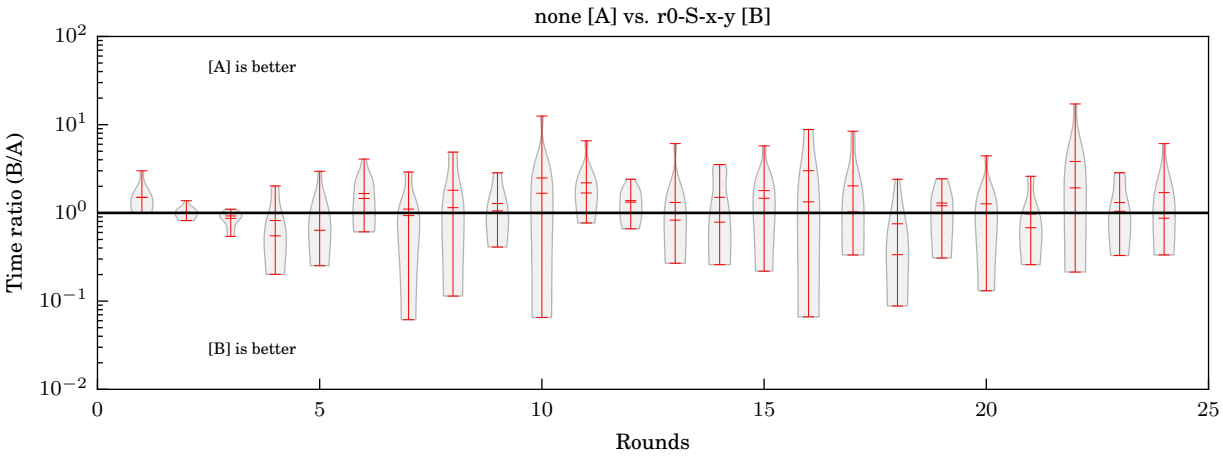

Figure 4.2: Violin plot showing the ratios distribution of solving time for the *none* and *r0-S-x-y* strategies.

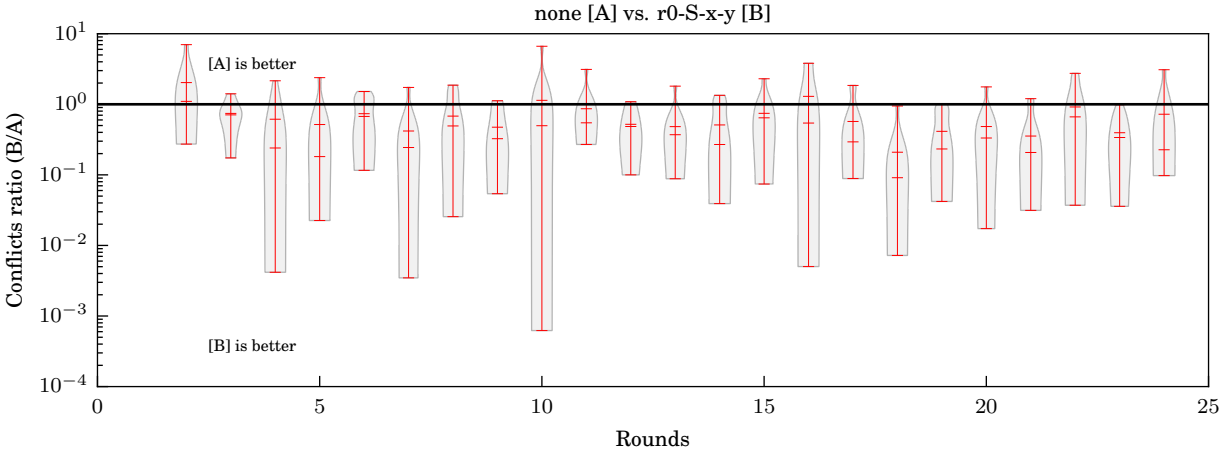

Figure 4.3: Violin plot showing the ratios distribution of number of conflicts for the *none* and *r0-S-x-y* strategies.

4.4 Branching order evaluation

For branching order optimization described in section 3.3 we again used the *SHA-1* and *SHA-3* hash functions for evaluation.

4.4.1 SHA-1 analysis

We tested this optimization on an 8-bit preimage attack on full 80 rounds. We compared the default MiniSat behavior to strategies where the variables corresponding to the value F of the round function were branched on first. We first tested all 80 possible rounds – that is, for every $0 \leq i < 80$ we tested a branching order strategy where the value of F from the i-th round was branched on first.

While with the default behavior these instances were solved in just a few seconds