Implementation of bit-vector variables in a constraint solver with an application to the generation of cryptographic substitution boxes

Master Thesis in Computer Science
at the Department of Information Technology
at Uppsala University
by Kellen Dye

August 29, 2014

Contents

Acknowledgements

I would like to thank my advisor Jean-Noël Monette, who proposed the subject of this thesis and who provided valuable suggestions and feedback as well as several crucial insights. Additionally, I would like to thank Pierre Flener for his excellent teaching in his constraint programming course at Uppsala University, and Christian Schulte for producing the most complete and clear definition of constraint programming I have read, and for his prompt help with issues on the Gecode mailing list.

Chapter 1

Introduction

Secure communications rely on cryptography in order to hide the contents of messages from eavesdroppers. On the internet, these messages might be users' passwords, bank details, or other sensitive information. For the communications to truly be secure, the cryptography must be strong enough to withstand attacks from dedicated adversaries.

Substitution boxes are a component in some cryptographic protocols which are critical to the strength of the entire system. These substitution boxes should have particular properties in order to ensure that the cryptographic system can withstand efforts to decrypt a message.

Substitution boxes are arrays of bit-vectors, where each bit-vector is itself an array of binary digits, or *bits* (a 0 or a 1). The desirable properties of a substitution box can be described as relationships between its constituent bit-vectors.

Constraint programming is a technique used to find values for variables in such a way that certain relationships between the variables are maintained and, optionally, the best values are found. If each bit-vector in a substitution box is represented by a variable, then constraint programming can be used to express the desirable properties of substitution boxes and then find substitution boxes which fulfill these properties.

Constraint programming typically occurs in the context of a program called a *constraint solver* which provides different types of variables such as integers or Booleans. Currently, most solvers do not have support for bit-vectors.

Although it is possible to convert the bit-vectors into another form (for example, interpreting each bit-vector as a number of Boolean variables; one for each bit), some of the desirable properties of substitution boxes are much more easily expressed in terms of bit-vectors.

This thesis therefore describes the implementation of bit-vector variables in the open-source constraint solver Gecode and their application to the problem of finding high-quality cryptographic substitution boxes.

In Chapter 2, constraint programming is defined, bit-vectors and operations

over bit-vectors are introduced, and substitution boxes and their desirable properties are described.

In Chapter 3, the works on which this thesis is based are reviewed. Michel and Van Hentenryck introduce bit-vector variables and domains for constraint programming and also define a number of propagators for bit-vector operations [9]. Ramamoorthy et al. suggested the application of constraint programming to substitution box generation [12] as well as some method for breaking symmetry in the search space [11].

In Chapter 4 we present our contributions: two additional symmetries of substitution boxes, several additional bit-vector propagators, an addition to the non-linearity constraint, and a correction to the S-7 constraint presented by Ramamoorthy which invalidates previous results.

In Chapter 5, we detail the bit-vector variable implementation in Gecode.

In Chapter 6, we present several alternative bit-vector models for substitution box generation, then in Chapter 7, we describe a set variable-based model and a Boolean variable-based model. We also define new global propagators for the S-2 nonlinearity constraint and the S-7 constraint.

In Chapter 8, we give a comparison of the various models for substitution box generation and describe their relative merits. Experimental evaluation indicates that modeling substitution boxes with bit-vector variables is an improvement over both set-based model and Boolean-based models and that that efficiency can be improved by the implementation of global propagators for some of the substitution box criteria.

Finally, in Chapter 9 we summarize the thesis and present possibilities for future work.

Chapter 2

Background

Here we present an introduction to constraint programming, bit-vectors, and substitution boxes. We use sudoku as a simple example problem for constraint programming, then present a more formal definition. Bit-vectors and the relevant operations over them are then presented. Finally, substitution boxes and their desirable properties are introduced.

2.1 Constraint Programming

Constraint programming is a method of solving problems in which the problem is characterized as a set of *constraints* over a set of *variables*, each of which has a *domain* of potential values. The constraints describe relationships between variables and limits on the variable domains. Taken together, these form a *constraint satisfaction problem*.

Finding a solution to a specific problem is typically done within a *constraint solver*, a program which can be used for many different problems and which can provide implementations for commonly used constraints.

The solver performs inference on the potential domains of the variables in a process called *propagation* in which the domain of a variable is reduced by eliminating values not satisfying a constraint and subsequently applying the implications of the altered domain to other variables. Once the propagators can no longer make additional changes to variable domains, they are said to be at *fixpoint* and the solver then performs systematic *search* by altering the domain of a variable. Search is interleaved with the propagation described above.

Search occurs in the *solution space*, that is, all possible combinations of values for all variables. The searching of the solution space can be represented as a *search tree*, where each search choice creates a branch and propagation occurs at the nodes of the tree. In the case that a search choice leads to the violation of a constraint or to a variable having no potential values, that part of the space is said to be *failed* and the search engine backtracks up the search tree and tries a different choice.

	2		5		1		9	
8			2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Figure 2.1: An example sudoku puzzle

Once a variable's domain is reduced to a single value, it is *assigned*, and once all variables are assigned the solver has produced a *solution*, so long as no constraints have been violated.

Certain problems may also require a problem-specific *objective function* to be optimized. The objective function evaluates the variable domains and produces a value which is to be minimized or maximized. Problems with such an objective function are called *constrained optimization problems*.

Constraint programming follows a *declarative* programming paradigm. Most programming is *imperative*, in which the steps needed to produce a solution are written by the programmer and followed by the computer. In declarative programming, the programmer describes the properties of a problem's solution, but not the method by which this should be computed [5].

2.1.1 Example: Sudoku

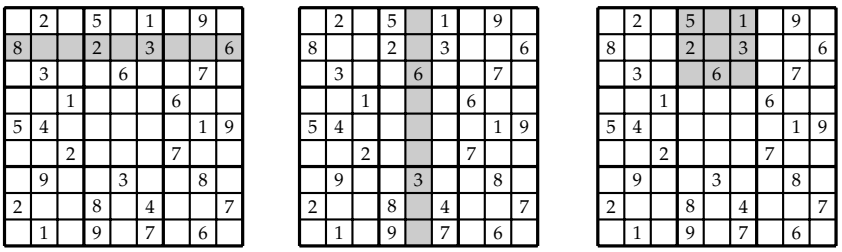
Sudoku is a kind of combinatorial puzzle which provides a simple example for demonstrating the application of constraint programming to a concrete problem. See Figure 2.1 for an example sudoku puzzle.

The goal of sudoku is for the player to fill in each of the grid squares of a 9×9 grid with one the numbers 1–9. Some grid squares are pre-filled and are intended to give a unique solution to the puzzle. The sudoku grid is divided into rows (Figure 2.2a) and columns (Figure 2.2b), and into nine 3×3 blocks (Figure 2.2c).

The rules of sudoku are simple: in each row, column, or block, each of the numbers 1–9 must be present and may not be repeated. Once all grid squares are filled and this rule is fulfilled, the puzzle is solved.

To produce a constraint satisfaction problem from a specific sudoku puzzle, the puzzle must first be *modelled* by defining the variables, their domains, and the constraints imposed upon them.

Each grid square can be represented by a variable, so there are $9 \times 9 = 81$ variables. Each of these variables can take on values 1–9, so their initial domains are: $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$.



(a) row (b) column (c) block

Figure 2.2: Regions of the sudoku grid

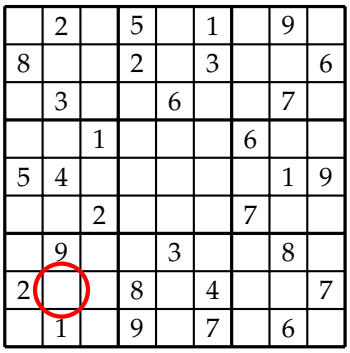


Figure 2.3: The grid square represented by x

For the pre-filled values, the domain associated with that variable contains only a single value, for example: $\{5\}$

The rules of the puzzle are implemented as constraints over a subset of the variables. The rule we wish to express is that the values taken by the variables in a given row (or column, or block) should all be different. The constraint which can enforce this relationship is also called alldifferent. A constraint must be defined for each row, column and block in the grid so a complete model consists of $9\ rows + 9\ columns + 9\ blocks = 27\ constraints$

Once the variables, domains, and constraints are defined, they may be used by a constraint solver to solve the problem by first reducing the variable domains by propagation, then by search, if necessary.

Propagation for a single variable

In Figure 2.3, a single grid square is indicated; let it be represented by a variable x. Initially, the domain for x is: $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, like the domains of all variables representing empty grid squares.

The variable x is a member of three alldifferent constraints: one for its block, one for its column, and one for its row.

First we apply the alldifferent constraint for the block; in the block there are the assigned values 1, 2, and 9. Since the constraint says that x's value must

be different from all other values, we remove these from the domain of x: $\{3, 4, 5, 6, 7, 8\}$

Next, the alldifferent constraint for the column is applied; here there are the values 1, 2, 3, 4, and 9. The values 1, 2, and 9 are no longer a part of x's domain, so these have no effect. We remove the remaining values 3 and 4 from the domain of x: $\{5, 6, 7, 8\}$

Finally, the alldifferent constraint for the row is applied. The values 2 and 4 have already been removed, but the remaining values 7 and 8 are removed from the domain of x: $\{5, 6\}$

Thus, by propagating just the three constraints in which x is directly involved and only examining the initially-provided values, the domain of x has been reduced to only two possibilities. Actual propagation by a constraint solver will also change the domains of the other variables, and provide better inference. For example, if x's domain is $\{5, 6\}$, but no other variable in its block has 5 in its domain, the propagator can assign x to 5 directly.

2.1.2 Definition

In the following definition of constraint programming we follow the terminology and format of the presentation of constraint programming given by Schulte in [13].

A constraint satisfaction problem (CSP) is a triple $\langle V, U, C \rangle$ where V is a finite set of variables, U is a finite set of values, and C is a finite set of constraints. Each constraint $c \in C$ is a pair $\langle v, s \rangle$ where v are the variables of c, $v \in V^n$, and s are the solutions of c, $s \subseteq U^n$ for the arity n, $n \in \mathbb{N}$.

For notational convenience, the variables of c can be written $\mathsf{var}(c)$ and the solutions of c can be written $\mathsf{sol}(c)$.

An assignment a is a function from the set of variables V to a universe U: $a \in V \to U$. For the set of variables $V = \{x_1, \ldots, x_k\}$, a particular variable x_i is assigned to the value n_i, that is: $a(x_i) = n_i$.

A particular assignment a is a solution of constraint c, written $a \in c$, if for $\mathsf{var}(c) = \langle x_1, \ldots, x_n \rangle$, $\langle a(x_1), \ldots, a(x_n) \rangle \in \mathsf{sol}(c)$

The set of solutions to a constraint satisfaction problem $\mathscr{C} = \langle V, U, C \rangle$ is

$$\mathsf{sol}(\mathscr{C}) = \{a \in V \to U \mid \forall c \in C : a \in c\}$$

Propagation

In constraint solvers, constraints are implemented with *propagators*. Propagators perform inference on sets of possible variable values, called constraint stores, s, with $s \in V \to 2^U$, where 2^U is the power set of U. The set of all constraint stores S is $S = V \to 2^U$.

If, for two stores s_1 and s_2, $\forall x \in V : s_1(x) \subseteq s_2(x)$, then s_1 is *stronger* than s_2, written $s_1 \leq s_2$. s_1 is *strictly stronger* than s_2, written $s_1 < s_2$, if $\exists x \in V : s_1(x) \subset s_2(x)$

Algorithm 2.1 Propagation algorithm

function propagate($\langle V, U, P\rangle, s$)
  $Q \leftarrow P$
  while $Q \neq \emptyset$ **do**
    $p \leftarrow \mathsf{select}(Q)$
    $\langle m, s'\rangle \leftarrow p(s)$
    $Q \leftarrow Q - \{p\}$
    if $m =$ *subsumed* **then**
      $P \leftarrow P - \{p\}$
    $MV \leftarrow \{x \in V \mid s(x) \neq s'(x)\}$
    $DP \leftarrow \{p \in P \mid \mathsf{var}(p) \cap MV \neq \emptyset\}$
    if $m =$ *fix* **then**
      $DP \leftarrow DP - \{p\}$
    $Q \leftarrow Q \cup DP$
    $s \leftarrow s'$
  return $\langle P, s\rangle$
end function

If $\forall x \in V : a(x) \in s(x)$ for an assignment a and a store s, then a is *contained in the store*, written $a \in s$. The store $\mathsf{store}(a) \in V \to 2^U$ is defined as: $\forall x \in X : \mathsf{store}(a)(x) = \{a(x)\}$

A propagator p is a function from constraint stores to constraint stores: $p \in S \to S$. A propagator must be *contracting*, $\forall s \in S : p(s) \leq s$ and *monotonic*, $\forall s_1, s_2 \in S : s_1 \leq s_2 \implies p(s_1) \leq p(s_2)$.

A store s is *failed* if $\exists x \in V : s(x) = \emptyset$. A propagator p fails on a store s if $p(s)$ is failed.

A constraint model is $\mathscr{M} = \langle V, U, P\rangle$, where V and U are the same as defined previously and P is a finite set of propagators over V and U.

The set of solutions of a propagator p is

$$\mathsf{sol}(p) = \{a \mid a \in V \to U, p(\mathsf{store}(a)) = \mathsf{store}(a)\}$$

The set of solutions for a particular model $\mathscr{M}$ is $\mathsf{sol}(\mathscr{M}) = \bigcap_{p \in P} \mathsf{sol}(p)$

For a propagator $p \in S \to S$, the variables involved in the particular constraint implemented by p are given by $\mathsf{var}(p)$.

The *fixpoint* of a function f, $f \in X \to X$, is an input x, $x \in X$ such that $f(x) = x$. A propagator p is at fixpoint for a store s if $p(s) = s$. A propagator p is *subsumed* by a store s if $\forall s' \leq s : p(s') = s'$, that is if all stronger stores are fixpoints of p.

In order to implement propagation in a solver, it is useful for the propagators to be able to report about the status of the returned store. We therefore define an *extended propagator*, ep, as a function from constraint stores to a pair containing a status message and a constraint store: $ep \in S \to SM \times S$ where the status message $SM = \{nofix, fix, subsumed\}$.

The application of an extended propagator ep to a constraint store s results in a tuple $\langle m, s' \rangle$, where the status message m indicates whether s' is a fixpoint ($m = \mathit{fix}$), whether s' subsumes ep ($m = \mathit{subsumed}$), or if no information is available ($m = \mathit{nofix}$).

A propagation algorithm using extended propagators is given in Algorithm 2.1. Initially, the solver will schedule all propagators P in the queue Q. The propagation algorithm executes the following inner loop until the queue is empty.

One propagator, p, is selected by a function select, which is usually specified dependent upon the problem to be solved. The propagator p is executed, then its returned status, m, is examined. If p is subsumed by the returned store s', then p is removed from P, since further executions will never reduce variable domains.

Next, the set of *modified variables*, MV, is calculated. From this set, the set of propagators which are dependent upon these variables, DP, is calculated. If s' is a fixpoint of p, then p is removed from the list of dependent propagators.

Finally, the dependent propagators are added to the queue if they were not already present, and the current store s is updated to be the store s', the result of executing p. In this step, changes to variable domains (indicated by MV) cause propagators to be scheduled for execution (added to Q), thus *propagating* these changes to other variable domains.

Once the queue Q is empty, propagation is complete and the set of non-subsumed propagators P and the updated store s are returned.

Search

A branching for $\mathscr{M}$ is a function b which takes a set of propagators Q and a store s and returns an n-tuple $\langle Q_1, \ldots, Q_n \rangle$ of sets of propagators Q_i.

A search tree for a model $\mathscr{M}$ and a branching b is a tree where the nodes are labelled with pairs $\langle Q, s \rangle$ where Q is a set of propagators and s is a store obtained by constraint propagation with respect to Q.

The root of the tree is $\langle P, s \rangle$, where $s = \mathsf{propagate}(P, s_{init})$ and $s_{init} = \lambda x \in V.U$. Each leaf of the tree, $\langle Q, s \rangle$, either has a store s which is failed or at which $b(Q, s) = \langle\rangle$ in which case the leaf is *solved*. For each inner node of the tree, s is not failed and $b(Q, s) = \langle Q_1, \ldots, Q_n \rangle$ where $n \geq 1$. Each inner node has n children where each child is labelled $\langle Q \cup Q_i, \mathsf{propagate}(Q \cup Q_i, s) \rangle$ for $1 \leq i \leq n$.

To actually construct a search tree (that is, to search the solution space) some exploration strategy must be chosen. As an example, a depth-first exploration procedure is given in Algorithm 2.2.

Algorithm 2.2 Depth-first exploration

```
function dfe(P, s)
    s' ← propagate(P, s)
    if s' is failed then
        return s'
    else
        case b(P, s')
        of ⟨⟩ then
            return s'
        of ⟨P1, P2⟩ then
            s'' ← dfe(P ∪ P1, s')
            if s'' is failed then
                return dfe(P ∪ P2, s')
            else
                return s''
end function
```

2.2 Bit-vectors

Bit-vectors are arrays of Boolean variables where each variable is represented by a single binary digit or *bit*.

The bits of an n-bit bit-vector x are addressed as x_i where i is the *index*, with $0 \leq i < n$. The bit at index 0 is the *least significant bit* while the bit at $n-1$ is the *most significant bit*.

When a bit-vector is written as a binary number or string, the least significant bit is written at the right; for example, if $x = 1110$, the least significant bit, x_0, is 0.

2.2.1 Integers

A bit-vector x can be interpreted as a non-negative integer:

$$I(x) = \sum_{i=0}^{n-1} x_i \cdot 2^i$$

2.2.2 Concatenation

A bit-vector can be seen as a concatenation of bits:

$$x = x_0 \parallel x_1 \parallel \ldots \parallel x_{n-1} = \mathop{\Big\|}_{i=0}^{n-1} x_i$$

So the concatenation of a bit-vector x and a single bit b is:

$$x \parallel b = x_0 \parallel x_1 \parallel \ldots \parallel x_{n-1} \parallel b \qquad \text{or} \qquad b \parallel x = b \parallel x_0 \parallel x_1 \parallel \ldots \parallel x_{n-1}$$

Note that the argument to the right of the concatenation operator is concatenated after the most significant bit of the bit-vector, that is, to the *left* side of the bit-vector, when written as a string.

For example, the bit-vector 100 could be written as $0 \parallel 0 \parallel 1$, and the concatenation of the bit-vector 100 and the bit 1, $100 \parallel 1$ results in the bit-vector 1100.

2.2.3 Bitwise operations

Bitwise operations are logical Boolean operations applied to each bit position of the bit-vector inputs to the operation. For operations with more than one input, the inputs must be of the same length, n. The result of a bitwise operation is a bit-vector also of length n.

The result of bitwise operations can therefore be seen as concatenations of the logical operations applied to single bit positions. For a binary Boolean operation op and input bit-vectors x and y, both of length n:

$$x \; op \; y = \prod_{i=0}^{n-1} (x_i \; op \; y_i)$$

And for unary operations:

$$op \; x = \prod_{i=0}^{n-1} (op \; x_i)$$

The bitwise operations are denoted with their logical operator symbols or name: AND ($\wedge$), OR ($\vee$), XOR ($\oplus$), NOT ($\neg$).

2.2.4 Parity

The result of the Boolean function *parity* is 1 if the number of ones in the input bit-vector x is odd and 0 otherwise:

$$\mathsf{parity}(x) = x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1} = \bigoplus_{i=0}^{n-1} x_i$$

2.2.5 Linear combination

The linear combination of two bit-vectors x and y is the parity of the bitwise AND:

$$\begin{aligned} \mathsf{linear}\,(x, y) &= (x_0 \wedge y_0) \oplus (x_1 \wedge y_1) \oplus \ldots \oplus (x_{n-1} \wedge y_{n-1}) \\ &= \bigoplus_{i=0}^{n-1} (x_i \wedge y_i) \\ &= \mathsf{parity}\,(x \wedge y) \end{aligned}$$

2.2.6 Hamming weight

The hamming weight is the integer count of the nonzero bits of a bit-vector x:

$$\mathsf{weight}(x) = x_0 + x_1 + \ldots + x_{n-1} = \sum_{i=0}^{n-1} x_i$$

2.2.7 Hardware support

Computers typically represent data as bit-vectors in both CPU registers and in memory. The CPU operates on fixed-length bit-vectors, called words, which can differ in length from machine to machine [19]. Bitwise operations and shifts are provided as CPU instructions in, for example, x86 processors [3].

2.2.8 Support in constraint solvers

Constraint solvers typically reason over integer variables, but additional variable types have been implemented. The bit-vectors described in this thesis are implemented in Gecode, an open-source constraint solver, written in C++ [6]. Gecode currently provides integer, Boolean, float, and set variables [17].

In some cases it can be more natural to reason about bit-vectors than other representations. As an example, the XOR operation is more-easily understood using bit-vector rather than integer variables.

Constraints which can be expressed with bit-vectors can of course also be expressed using arrays of Boolean variables with one variable per bit, or with set variables where the set contains the indexes of the 'on' bits. The approach with an array of Boolean variables is called *bit-blasting* which can create very large numbers of variables.

Using bit-vector variables in a constraint solver allows for constant-time propagation algorithms if the length of the bit-vectors is less than the word size of the underlying computer architecture [9].

2.3 Substitution boxes

The goal of cryptography is to hide the contents of messages such that if a message between two parties is intercepted by a third, this third party will not easily be able to recover the message contents.

S_4		Middle 4 bits															
		0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Outer bits	00	0111	1101	1110	0011	0000	0110	1001	1010	0001	0010	1000	0101	1011	1100	0100	1111
	01	1101	1000	1011	0101	0110	1111	0000	0011	0100	0111	0010	1100	0001	1010	1110	1001
	10	1010	0110	1001	0000	1100	1011	0111	1101	1111	0001	0011	1110	0101	0010	1000	0100
	11	0011	1111	0000	0110	1010	0001	1101	1000	1001	0100	0101	1011	1100	0111	0010	1110

Figure 2.4: DES's 6×4 S-box S_4. Highlighted is the row and column for the input pattern 110000. The output for this pattern is 1111.

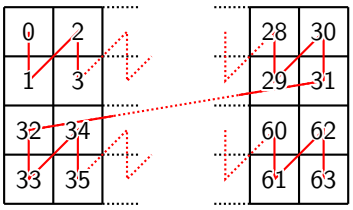


Figure 2.5: The pattern of inputs in a 6×4 S-box

In order for both parties to be able to read messages from the other, they must have some kind of shared secret information. One such method is for both parties to have a copy of the same *key*, which both encrypts ("locks") and decrypts ("unlocks") the contents of a message; this is called *symmetric-key cryptography*. The method or algorithm by which encryption is performed is called a *cipher* and the encrypted text produced by a cipher is called the *ciphertext*.

Substitution boxes, or *S-boxes*, are used in order to obscure the relationship between the key and the ciphertext, a property called *confusion* [18]. S-boxes are look-up tables which provide a mapping from a certain input to an output. An example of an S-box is given in Figure 2.4.

S-boxes are categorized based on their input and output sizes; a box which takes n bits of input and produces m bits of output is a $n \times m$ S-box. The example given is a 6×4 S-box. The input for the example S-box is 6 bits long, the first and last of which are used to determine the row, while the middle 4 bits determine the column. In the example, an input of 110000 results in an output of 1111. Figure 2.5 shows the pattern made by increasing inputs to an S-box; for an even input i, the next input, $i + 1$ is on the subsequent row, and for an odd input j, the next input, $j + 1$ is on the preceding row, except for $j = 2^{n-1} - 1$, which is the last input in its row.

Substitution boxes are used in substitution-permutation networks and Feistel ciphers, both of which divide the encryption process into a number of *rounds* in which a portion of the message and a portion of the key are combined and then passed through a number of S-boxes to produce a new interim message which will be processed by further rounds of the encryption process [20]. The Data Encryption Standard, or DES, is a once-popular symmetric-key encryption scheme which uses a Feistel cipher.

DES operates on a 64-bit block of a message and produces a 64-bit ciphertext as output. Each block is passed through 16 rounds, each of which operates on a 32-bit half of a message block and a 48-bit subkey. A different subkey is generated for each round from the overall key. See Figure 2.6.

In each DES round, shown in Figure 2.7, the half-block is first expanded to 48-bits, then the expanded half-block and the subkey are XORed together. The resultant 48-bit block is then passed through 8 predefined S-boxes, each of which takes 6 bits of input and gives a 4-bit output, resulting in a new 32-bit interim message. The interim message is then *permuted* by a *permutation box* or *P-box* which *diffuses* the contents of the message in order to make the output more uniform and therefore more difficult to analyze. Finally, the result of the

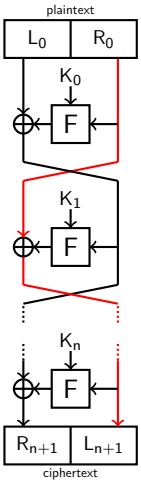


Figure 2.6: Feistel cipher. The 64-bit plaintext is initially split into two 32-bit half-blocks, L_0 and R_0, and passed into the *Feistel function* F, along with the subkey for round 0, K_0. Each subsequent round follows the same pattern.

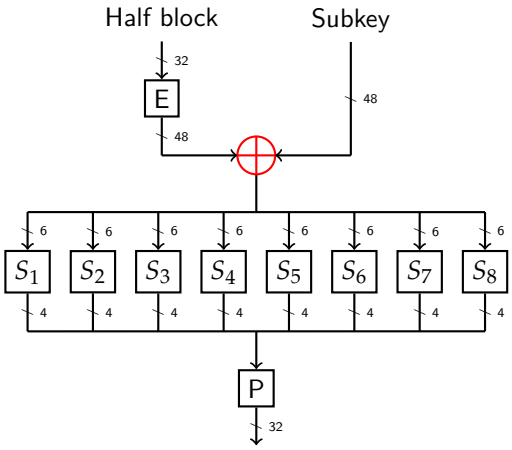


Figure 2.7: Feistel function, F. The 32-bit half-block passes through the extender E to produce a 48-bit extended block. The extended block is XORed with the 48-bit subkey, then split into eight 6-bit portions which are passed through the substitution boxes S_1–S_8, producing eight 4-bit outputs. These outputs are reassembled into a single 32-bit pattern and finally passed through the permutation box P.

		β														
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	...								...							
	13	6	0	2	0	-2	4	-10	-2	0	-2	4	-2	8	-6	0
	14	-2	-2	0	-2	4	0	2	-2	0	4	2	-4	6	-2	-4
α	15	-2	-2	8	6	4	0	2	2	4	8	-2	8	-6	2	0
	16	2	-2	0	0	-2	-6	-8	0	-2	-2	-4	0	2	10	-20
	...								...							

Table 2.1: Portion of a linear approximation table for DES's S_5

P-box is XORed with the half-block which was used as input to the previous round in order to produce the new half-block for the following round [2].

As the S-boxes are the only non-linear part of a Feistel or substitution-permutation network, and it is the non-linearity of the system which makes it difficult to cryptanalyze, it is critical to the security of the system that the S-boxes be as non-linear as possible [12]. In the next section, we describe ways of evaluating the linearity of an S-box.

2.3.1 Measuring the linearity of an S-box

The linearity of an S-box can be investigated by examining the correlation between the input and output bits of an $n \times m$ S-box S.

Matsui [8] calculates the probability that a set of input bits α (a bit-vector) coincides with a set of output bits β (another bit-vector). A desirable property for an S-box is that this probability, $p(\alpha, \beta)$, should be close to $1/2$.

For an S-box S, the number of matches between a pair α and β is the count of the number of times $\mathsf{linear}(x, \alpha)$ equals $\mathsf{linear}(S(x), \beta)$ for all possible inputs x:

$$N(\alpha, \beta) = |\{x \mid 0 \leq x < 2^n, \mathsf{linear}(x, \alpha) = \mathsf{linear}(S(x), \beta)\}|$$

where n is the number of input bits to the S-box and $S(x)$ is the output of the S-box S for the input x. We will call $N(\alpha, \beta)$ the *count* of the S-box, as Matsui did not give a name.

Since $0 \leq N(\alpha, \beta) \leq 2^n$, the probability that some α coincides with some β is

$$p(\alpha, \beta) = \frac{N(\alpha, \beta)}{2^n}$$

Equivalently, since $p(\alpha, \beta)$ should be close to $1/2$, $N(\alpha, \beta)$ should be close to $2^n/2$. In the case of the 6×4 S-boxes used in DES, $N(\alpha, \beta)$ should be close to 32.

Example 2.1. *For a 6×4 S-box and $\alpha = 16$ (binary 010000), $\beta = 15$ (binary 1111), the value $N(16, 15) = 12$ indicates that the probability that an input bit at index 5 of S^1 coincides with an XORed value of all output bits with probability $12/64 \approx 0.19$.*

$N(\alpha, \beta)$ itself is less interesting than its closeness to $1/2$ which Matsui collects in a table called the *linear approximation table*, *LAT*. A portion of the *LAT* for DES's

[1] Matsui's example; he gives this as the fourth input bit, presumably using 0-based ordinals

S_5 substitution box is given in table 2.1. The rows of the table correspond to α, where $1 \leq \alpha < 2^n$, and the columns correspond to β where $1 \leq \beta < 2^m$.

An entry in this table is

$$LAT(\alpha, \beta) = N(\alpha, \beta) - \frac{2^n}{2}$$

An S-box is only as good as its weakest point, so an overall evaluation criteria for non-linearity can be defined as a *score*, σ, where "good" S-boxes have a lower score:

$$\sigma = \max_{\alpha,\beta} \{|LAT(\alpha, \beta)|\}$$

2.3.2 DES S-Box design criteria

DES was designed at IBM in the 1970s, but the design criteria for the substitution boxes were kept a secret until after the description of differential cryptanalysis by Biham and Shamir in [2]. In 1994, Coppersmith disclosed the criteria in [4], given in Table 2.2.

Criteria S-2 describes a weaker evaluation of an S-box for linearity than the one given by Matsui. Coppersmith calls Matsui's variant S-2′ and recommends it as a better criteria for the evaluation of future cryptographic systems. Specifically, S-2 says only that *single output bits* should not be "too close to a linear function of the input bits" while S-2′ considers all possible combinations of output bits.

These criteria form the basis of the constraint programming model for substitution box generation discussed in the next chapter. Note that criteria S-4, S-5, and S-6 are requirements on pairs of S-box entries; these can be enforced by simple pairwise constraints. In contrast, criteria S-2, S-3, and S-7 are requirements on groups of variables; these can be enforced by global propagators which can potentially achieve much better propagation than a collection of simple constraints expressing the same requirement.

S-1 Each S-box has six bits of input and four bits of output.

S-2 No output bit of an S-box should be too close to a linear function of the input bits.

S-3 If we fix the leftmost and rightmost input bits of the S-box and vary the four middle bits, each possible 4-bit output is attained exactly once as the middle four input bits range over their 16 possibilities.

S-4 If two inputs to an S-box differ in exactly one bit, the outputs must differ in at least two bits.

S-5 If two inputs to an S-box differ in the two middle bits exactly, the outputs must differ in at least two bits.

S-6 If two inputs to an S-box differ in their first two bits and are identical in their last two bits, the two outputs must not be the same.

S-7 For any nonzero 6-bit difference between inputs, ΔI, no more than eight of the 32 pairs of inputs exhibiting ΔI may result in the same output difference ΔO

Table 2.2: DES design criteria

Chapter 3

Previous work

In this chapter the works on which this thesis is based are presented. We first introduce a bit-vector variable domain and bit-vector constraints described by Michel and Van Hentenryck in [9]. We then review two works by Ramamoorthy et al. which describe the application of constraint programming to S-box generation and symmetries of S-boxes, based on the DES design criteria [12][11].

3.1 Bit-vector variables & constraints

Ordering on bit-vectors b_1 and b_2 is defined according to the integer representation: $b_1 \leq b_2$ if $I(b_1) \leq I(b_2)$.

A bit-vector domain is a pair $\langle l, u \rangle$ where l and u are bit-vectors and $l_i \leq u_i$ for $0 \leq i < k$. The *free bits* of the domain, the bits which are not assigned to either 1 or 0, are

$$V(\langle l, u \rangle) = \{i \mid 0 \leq i < k, l_i < u_i\}$$

The free bits can be found in constant time

$$\mathsf{free}(\langle l, u \rangle) = u \oplus l$$

The *fixed bits* of the domain, the bits which are assigned to either 1 or 0, are

$$F(\langle l, u \rangle) = \{i \mid 0 \leq i < k, l_i = u_i\}$$

The fixed bits can be found in constant time

$$\mathsf{fixed}(\langle l, u \rangle) = \neg\, \mathsf{free}(l, u)$$

A bit-vector domain then represents the set of bit-vectors

$$\{b \mid l \leq b \leq u \wedge \forall i \in F(\langle l, u \rangle) : b_i = l_i\}$$

A bit domain is the domain of bit i in a bit-vector domain D, $D_i = \{b_i \mid b \in D\}$.

A bit-vector variable x is associated with a domain $D = \langle l, u \rangle$. l^x and u^x denote the bit-vectors defining x's domain and F^x and V^x denote $F(\langle l^x, u^x \rangle)$ and $V(\langle l^x, u^x \rangle)$.

For a binary constraint c over two variables x and y with domains $\langle l^x, u^x \rangle$ and $\langle l^y, u^y \rangle$, the goal of propagation is to determine new domains $\langle low^x, up^x \rangle$ and $\langle low^y, up^y \rangle$ and to check if the constraint has failed. The check for failure is done by checking if the new domains are valid, that is if $\forall i : 0 \leq i < k : l_i \leq u_i$. Assuming that the length of the bit-vector is less than the length of the word-size of the executing computer, this test can be done in constant time by the expression

$$\mathsf{valid}(\langle l, u \rangle) = \neg(l \oplus u) \vee u$$

where valid is true if all bits of the resultant bit-vector are set to 1.

Michel and Van Hentenryck define propagators for a number of bit-vector operations; described here are only the propagators for the constraints used in the implementation of the DES design criteria.

The propagator for equality between two bit-vector variables x and y is given in Algorithm 3.1.

Algorithm 3.1 Propagator for $x = y$

function propagate($x = y$)
  $low^x \leftarrow l^x \vee l^y$
  $low^y \leftarrow low^x$
  $up^x \leftarrow u^x \wedge u^y$
  $up^y \leftarrow up^x$
  $\langle l^x, u^x \rangle \leftarrow \langle low^x, up^x \rangle$
  $\langle l^y, u^y \rangle \leftarrow \langle low^y, up^y \rangle$
  return valid($\langle l^x, u^x \rangle$) $\wedge$ valid($\langle l^y, u^y \rangle$)
end function

The XOR constraint holds for three bit-vectors x, y, and z, of equal length if $x \oplus y = z$, the propagator for this relationship is given in Algorithm 3.2.

Algorithm 3.2 Propagator for $x \oplus y = z$

function propagate($x \oplus y = z$)
  $low^x \leftarrow l^x \vee (\neg u^z \wedge l^y) \vee (l^z \wedge \neg u^y)$
  $up^x \leftarrow u^x \wedge (u^z \vee u^y) \wedge \neg(l^y \wedge l^z)$
  $low^y \leftarrow l^y \vee (\neg u^z \wedge l^x) \vee (l^z \wedge \neg u^x)$
  $up^y \leftarrow u^y \wedge (u^z \vee u^x) \wedge \neg(l^x \wedge l^z)$
  $low^z \leftarrow l^z \vee (\neg u^x \wedge l^y) \vee (l^x \wedge \neg u^y)$
  $up^z \leftarrow u^z \wedge (u^x \vee u^y) \wedge \neg(l^x \wedge l^y)$
  $\langle l^x, u^x \rangle \leftarrow \langle low^x, up^x \rangle$
  $\langle l^y, u^y \rangle \leftarrow \langle low^y, up^y \rangle$
  $\langle l^z, u^z \rangle \leftarrow \langle low^z, up^z \rangle$
  return valid($\langle l^x, u^x \rangle$) $\wedge$ valid($\langle l^y, u^y \rangle$) $\wedge$ valid($\langle l^z, u^z \rangle$)
end function

The AND constraint holds for three bit-vectors x, y, and z, of equal length if $x \wedge y = z$, the propagator for this relationship is given in Algorithm 3.3.

Algorithm 3.3 Propagator for $x \wedge y = z$

function propagate($x \wedge y = z$)
  $up^x \leftarrow u^x \wedge (\neg((\neg u^z) \wedge l^y))$
  $low^x \leftarrow l^x \vee l^z$
  $up^y \leftarrow u^y \wedge (\neg((\neg u^z) \wedge l^x))$
  $low^y \leftarrow l^y \vee l^z$
  $up^z \leftarrow u^z \wedge u^x \wedge u^y$
  $low^z \leftarrow l^z \vee (l^x \wedge l^y)$
  $\langle l^x, u^x \rangle \leftarrow \langle low^x, up^x \rangle$
  $\langle l^y, u^y \rangle \leftarrow \langle low^y, up^y \rangle$
  $\langle l^z, u^z \rangle \leftarrow \langle low^z, up^z \rangle$
  return valid($\langle l^x, u^x \rangle$) $\wedge$ valid($\langle l^y, u^y \rangle$) $\wedge$ valid($\langle l^z, u^z \rangle$)
end function

The membership constraint, $x \in [L, U]$ holds for a bit-vector variable x and integers L and U if $L \leq I(x) \leq U$; the propagator for the membership constraint is given in Algorithm 3.4.

Algorithm 3.4 Propagator for $x \in [L, U]$

function propagate($x \in [L, U]$)
  $i \leftarrow k - 1$
  while $i \geq 0$ **do**
    if $I(u^x) < L \vee I(l^x) > U$ **then**
      return false
    if $i \in V^x$ **then**
      if $I(u^x) - 2^i < L$ **then**
        $l_i^x = 1$
      else if $I(l^x) + 2^i > U$ **then**
        $u_i^x = 0$
      else
        break
    $i \leftarrow i - 1$
  return true
end function

The constraint channel(x, X) for a bit-vector variable x and an interval variable X channels the domain bounds of x to X and vice-versa; a propagator for channeling is given in Algorithm 3.5.

Algorithm 3.5 Propagator for $\mathsf{channel}(x, X)$

```
function propagate(channel(x, X))
    if ¬ propagate(X ∈ [I(l^x), I(u^x)]) then
        return false
    if ¬ propagate(x ∈ D^X) then
        return false
    return propagate(X ∈ [I(l^x), I(u^x)])
end function
```

3.2 Constraint programming & substitution boxes

Ramamoorthy et al. propose the application of constraint programming to S-box generation, using the DES design criteria as a basis for their recommended constraints [12].

They represent the S-box as an array of 2^n decision variables, which fulfills criteria S-1. For criteria S-2, they introduce a series of heuristics for evaluating nonlinearity of S-boxes. They then define additional constraints for criteria S-3–S-7. For an S-box S, the decision variable corresponding to an input x is denoted $S(x)$.

3.2.1 Criteria S-2

To enforce criteria S-2, the non-linearity constraint, Ramamoorthy et al. initially propose a hard constraint, H_S, which rejects fully-assigned S-boxes whose score exceeds a threshold τ: $\sigma \leq \tau$

They then extend the idea of a threshold to S-boxes which have ϕ or more assigned variables. This "incomplete, incremental heuristic," H_I, allows the solver to fail a space where the score of the partially-filled S-box, σ_ϕ (defined below), exceeds the threshold τ: $\sigma_\phi \leq \tau$. Note that in the context of H_I, ϕ is a fixed value, so the heuristic does nothing until there are at least ϕ assigned variables.

Finally, they propose a "complete, incremental heuristic," H_C, which does not require a fixed number of assigned variables. They again use a threshold, τ, while the number of currently-assigned variables is given by ϕ:

$$\phi - \tau - \frac{2^n}{2} \leq \max_{\alpha,\beta} \left\{ N_\phi(\alpha, \beta) \right\} \leq \frac{2^n}{2} + \tau$$

This H_C heuristic requires that the weakest point of the S-box have a count which falls within the range of the threshold. The upper bound is unchanged from H_S and H_I, but the lower bound ensures that it is possible for the count of the partially-filled S-box to reach some part of the acceptable range given the remaining number of unassigned variables. $N_\phi(\alpha, \beta)$ adds a clause to the counting of matches between α and β, namely that $S(x)$ must be assigned:

$$N_\phi(\alpha, \beta) = |\{x \mid 0 \leq x < 2^n, \mathsf{assigned}\,(S\,(x)) \wedge \mathsf{linear}\,(x, \alpha) = \mathsf{linear}\,(S\,(x)\,, \beta)\}|$$

Recall that the count, $N(\alpha, \beta)$, has the range $0 \leq N(\alpha, \beta) \leq 2^n$ where n is the size of the S-box inputs. $N_\phi(\alpha, \beta)$ therefore has the range $0 \leq N_\phi(\alpha, \beta) \leq 2^\phi$.

The score for a partially-assigned S-box with ϕ assigned variables is then defined in terms of $N_\phi(\alpha, \beta)$ (omitting the linear approximation table, LAT):

$$\sigma_\phi = \max_{\alpha,\beta} \left\{ \left| N_\phi(\alpha, \beta) - \frac{2^n}{2} \right| \right\}$$

As given by Ramamoorthy, H_C is actually not complete; we extend the definition slightly in Section 4.2.

3.2.2 Criteria S-3

Criteria S-3 requires that the set of inputs with the same first and last bits must take on distinct values. The structure of the S-boxes are such that this is equivalent to saying that the variables in each row must take distinct values.

$$\forall i : 1 \leq i \leq 4 : \mathsf{alldifferent}(row_i)$$

3.2.3 Criteria S-4

Criteria S-4 requires that for any two inputs x and y which differ in exactly one bit, $S(x)$ and $S(y)$ must differ in at least two bits. The XOR ($\oplus$) operation results in a bit-vector indicating bits which differ, and the hamming weight applied to the resultant bit-vector gives a count of how many bits differ:

$$\begin{aligned} &\forall x : 0 \leq x < 2^n : \\ &\quad \forall y : x + 1 \leq y < 2^n : \\ &\qquad \mathsf{weight}\,(x \oplus y) = 1 \implies \mathsf{weight}\,(S(x) \oplus S(y)) \geq 2 \end{aligned}$$

3.2.4 Criteria S-5

Criteria S-5 requires that for any two inputs x and y which differ in exactly the two middle bits, $S(x)$ and $S(y)$ must differ in at least two bits. For a 6×4 S-box, the bit-vector with the two middle bits set is 001100

$$\begin{aligned} &\forall x : 0 \leq x < 2^n : \\ &\quad \forall y : x + 1 \leq y < 2^n : \\ &\qquad (x \oplus y) = 001100 \implies \mathsf{weight}(S(x) \oplus S(y)) \geq 2 \end{aligned}$$

3.2.5 Criteria S-6

Criteria S-6 requires that for any two x and y which differ in their first two bits but are the same in their last two, $S(x)$ and $S(y)$ must not be equal. The XOR ($\oplus$) operation over two bit-vectors gives the bits which differ. For a 6×4 S-box, applying a AND ($\wedge$) to the result of the XOR operation and the bit-vector 110011

isolates the first and last two bits; if the result is equal to 110000 then the input bit-vectors differ in their first two bits but are equal in their last two bits.

$$\begin{aligned}&\forall x : 0 \leq x < 2^n : \\ &\quad \forall y : x+1 \leq y < 2^n : \\ &\qquad ((x \oplus y) \wedge 110011) = 110000 \implies S(x) \neq S(y)\end{aligned}$$

Ramamoorthy gives two different values for the equality, $3 \cdot 2^{n-1}$ and $3 \cdot 2^{n-2}$; it is the second of these which is correct. For a 6×4 S-box, this is equivalent to the bit-vector 110000.

3.2.6 Criteria S-7

Unfortunately, it appears that Ramamoorthy et al. have misinterpreted the S-7 criteria. Specifically, they seem to have misinterpreted the phrase "any nonzero 6-bit difference" to mean that all six bits of inputs x and y must differ. A corrected constraint for S-7 is given in Section 4.3.

This misunderstanding invalidates the results given by Ramamoorthy et al.; they claim to have found a large number of 6×4 S-boxes with score 8, however, although the single instance of such a S-box given in their paper does have a score of 8, it violates the corrected S-7 constraint.

3.3 Symmetries

Ramamoorthy et al. also describe several symmetries of S-boxes and means to break these during search [11]. Based on analysis of constraints S-4, S-5, and S-6, they identify row, column, and diagonal symmetries.

The row symmetry allows the top two rows of the S-box to be exchanged so long as the bottom two are simultaneously exchanged. To break the row symmetry during search, the integer interpretation of the value for input 0 is constrained to be less than or equal to the integer interpretation of the value for the same position in the next row, namely input 1: $I(S(0)) \leq I(S(1))$

The column symmetry allows certain pairs of columns to be exchanged, as long as they are all simultaneously exchanged. Columns 0 and 6 are one such pair in a 6×4 S-box, so this symmetry can be broken during search by constraining the value for input 0 (which is in column 0) to be less than or equal to the value for input 12 (which is in column 6): $I(S(0)) \leq I(S(12))$

The diagonal symmetry allows quadrants of the S-box to be exchanged with the quadrant diagonal from it as long as all four quadrants are switched simultaneously. The top-left quadrant will therefore be exchanged with the lower-right quadrant; in a 6×4 S-box, the top-left value of the top-left quadrant is the value for input 0, while the top-left value of the bottom-right quadrant is the value for input 48. To break the diagonal symmetry during search, the integer representation of the value for input 0 is constrained to be less than or equal to the integer representation of the value for the input 48: $I(S(0)) \leq I(S(48))$

They then describe a rotational symmetry and a symmetry they call *bit inversion symmetry*. First, they establish some properties of bit-vectors and linear approximation tables. They show that the linear combination of the bitwise negation of a bit-vector x and a constant bit-vector y is

$$\mathsf{linear}(\neg x, y) = \mathsf{linear}(x, y) \oplus \mathsf{parity}(y)$$

Changing an assignment for an input x from $S(x)$ to $\neg S(x)$ changes the entry $LAT(\alpha, \beta)$ by

$$-1^{\mathsf{linear}(x,\alpha)\oplus\mathsf{linear}(\neg S(x),\beta)} \cdot \mathsf{parity}(\beta)$$

Swapping the assignments for the inputs x and $\neg x$ changes the entry $LAT(\alpha, \beta)$ by

$$\left((-1)^{\mathsf{linear}(x,\alpha)\oplus\mathsf{linear}(\neg S(x),\beta)} + (-1)^{\mathsf{linear}(\neg x,\alpha)\oplus\mathsf{linear}(\neg S(\neg x),\beta)}\right) \cdot \mathsf{parity}(\alpha)$$

Using these properties, they show that the score of a complete assignment remains unchanged if its assigned values are replaced by the NOT of those values; they call this *bit inversion symmetry*. To break this symmetry during search, the domain of the variable for input 0 is limited to the range 0–$(2^{m-1} - 1)$ (i.e., the most significant bit is set to 0). For a 6×4 S-box, this becomes: $I(S(0)) \leq 7$

They also show that the score of a complete assignment remains unchanged if the assigned values are swapped between the variables x and $\neg x$; this is equivalent to rotating a S-box by 180°. To break this symmetry during search, the variable for input 0 can be constrained to be less-than or equal-to the variable for input $2^n - 1$. For a 6×4 S-box, the integer representation of the value for input 0 is therefore constrained to be less than or equal to the integer representation of the value value for input 63: $I(S(0)) \leq I(S(63))$

Chapter 4

Contributions

In this chapter we describe several new bit-vector propagators for typical bit-vector operations, we present an addition to the H_C heuristic, we correct the S-7 constraint presented by Ramamoorthy, and finally we identify two new symmetries of S-boxes.

4.1 Propagators

Most of the bit-vector propagators implemented have already been described in Section 3.1. Some additional propagators are necessary for working with S-boxes but are nevertheless generally useful.

4.1.1 Hamming weight

The constraint $\mathsf{weight}(x) = y$ holds for a bit-vector variable x and a integer variable y when y is equal to the hamming weight of x; a propagation algorithm is given in Algorithm 4.1. For a bit-vector variable x, the possible hamming weights are an interval $[\mathit{min\_weight}, \mathit{max\_weight}]$.

The minimum weight of x, *min_weight*, is the weight of the currently-fixed bits of the bit-vector variable if the remaining free bits are fixed to zero. This is exactly the same as the lower bound, l^x, where all the free bits are represented as 0, while the fixed bits have their final values, so $\mathit{min\_weight} = \mathsf{weight}(l^x)$.

Conversely, for the maximum weight of x, *max_weight*: if all of the currently-free bits are fixed to 1, then x will be exactly equal to u^x, and $\mathsf{weight}(x) = \mathsf{weight}(u^x)$, so $\mathit{max\_weight} = \mathsf{weight}(u^x)$.

When the integer variable y is exactly equal to either the minimum or maximum weight of the bit-vector x, then x must be exactly l^x or u^x, respectively.

The complexity of the propagator for $\mathsf{weight}(x) = y$ itself is $\mathsf{O}(1)$; it needs only to compute the maximum and minimum weights of x, both of which can be done in constant time for bit-vector variables less than the word length of the underlying architecture. In our development environment, a 32-bit Linux using

gcc 4.7.2, weight is implemented by the standard library `bitset.count()`. Internally, `bitset` uses the gcc-built-in function `__builtin_popcount`, which on our system compiles to a single processor instruction `__popcountsi2`.

Algorithm 4.1 Propagator for $\mathsf{weight}(x) = y$

```
function propagate(weight(x) = y)
    min_weight ← weight(l^x)
    max_weight ← weight(u^x)
    if ¬ propagate(y ∈ [min_weight, max_weight]) then
        return false
    if assigned(y) then
        if y = max_weight then
            if ¬ propagate(x = u^x) then
                return false
        if y = mix_weight then
            if ¬ propagate(x = l^x) then
                return false
    return true
end function
```

4.1.2 Parity

The constraint $\mathsf{parity}(x) = y$ holds for a bit-vector variable x and a Boolean variable y when y is equal to the parity of x. Recall that $\mathsf{parity}(x)$ is true (1) when the number of set bits in x is odd and false (0) otherwise. A propagation algorithm is given in Algorithm 4.2.

It is possible to fix the value of the Boolean variable y only when x is assigned. In this case, y is propagated to the value *true* when $\mathsf{parity}(x) = 1$ and to *false* when $\mathsf{parity}(x) = 0$.

If y is assigned and x has only a single free bit, it is possible to fix this bit based on the value of y; if y is equal to the parity of the bits currently set to 1, then this final bit must be 0, otherwise, this final bit must be 1. The parity of the bits currently set to 1 is equivalent to the parity of the lower bound of x, $\mathsf{parity}(l^x)$.

The propagator for $\mathsf{parity}(x) = y$ is $\mathsf{O}(1)$; it must determine if a variable is assigned, it must find the free bits of a bit-vector variable, it must calculate the weight of a bit-vector, and it must calculate the parity of a bit-vector. We have previously described that the first three of these can be calculated in constant time.

The parity can be calculated by linear scan ($\mathsf{O}(n)$ on the number of bits n), by a "bit twiddling hack" [1] ($\mathsf{O}(1)$ with a look-up table), or by a `gcc`-builtin (also $\mathsf{O}(1)$, no look-up table necessary). In our development environment, parity is implemented using the `gcc`-built-in function `__builtin_parity`, which compiles down to a series of 8 assembly instructions.

Algorithm 4.2 Propagator for $\mathsf{parity}(x) = y$

```
function propagate(parity(x) = y)
    if assigned(x) then
        if parity(x) = 1 then
            if ¬ propagate(y = true) then
                return false
        else
            if ¬ propagate(y = false) then
                return false
    if assigned(y) ∧ ¬ assigned(x) then
        if weight(free(⟨l^x, u^x⟩)) = 1 then
            current_parity ← parity(l^x)
            if y = current_parity then
                if ¬ propagate(x = l^x) then
                    return false
            else
                if ¬ propagate(x = u^x) then
                    return false
    return true
end function
```

4.1.3 Disequality

The disequality constraint holds for a pair of bit-vectors, x and y, if $x \neq y$. A propagation algorithm is given in Algorithm 4.3.

For the constraint $x \neq y$ to hold, the values of the fixed bits which are common to both bit-vectors, *fixed_both*, must be unequal when x and y are both assigned. If the values of the common fixed bits becomes unequal at some point before both are assigned, the propagator may be subsumed. If both bit-vectors are assigned, but their common bits are equal, then $x = y$, so the propagator returns failure.

If one bit-vector is assigned while the other has a single free bit and the values of the common bits are equal, then the value of the final bit can be determined. If x is the assigned bit-vector, then the value of x for the free bit of y can be masked off with $l^x \wedge \mathsf{free}(\langle l^y, u^y \rangle)$. Inverting this value produces a mask such that $u^y \wedge \neg(l^x \wedge \mathsf{free}(\langle l^y, u^y \rangle))$ will be equivalent to y with the final free bit set to the inverse of the value of that bit in x.

The propagator for $x \neq y$ is also $\mathsf{O}(1)$. The propagator must check for assignment, find the free and fixed bits of bit-vector variables, calculate their hamming weight, and apply bitwise operations. We have previously presented constant-time operations for all of these.

Algorithm 4.3 Propagator for $x \neq y$

```
function propagate(x ≠ y)
    fixed_both ← fixed(l^x, u^x) ∧ fixed(l^y, u^y)
    if (l^x ∧ fixed_both) ≠ (l^y ∧ fixed_both) then
        return true
    if assigned(x) ∧ assigned(y) then
        return false
    if (assigned(x) ∧ ¬ assigned(y)) ∨ (¬ assigned(x) ∧ assigned(y)) then
        free_y ← free(⟨l^y, u^y⟩)
        if assigned(x) ∧ weight(free_y) = 1 then
            if ¬ propagate(y = u^y ∧ ¬(l^x ∧ free_y)) then
                return false
        free_x ← free(⟨l^x, u^x⟩)
        if assigned(y) ∧ weight(free_x) = 1 then
            if ¬ propagate(x = u^x ∧ ¬(l^y ∧ free_x)) then
                return false
    return true
end function
```

4.2 An extension to H_C

The goal of the H_C heuristic, described in Section 3.2.1, is to ensure that the overall score of the S-box is less than τ, but Ramamoorthy only defines H_C in terms of the maximum of $N_\phi(\alpha, \beta)$, omitting the minimum value, which can also be the cause of a high (bad) score. We therefore also define a lower bound which ensures that given ϕ assigned variables, it is possible for the minimum value of $N_\phi(\alpha, \beta)$ to reach the acceptable range once all values become assigned:

$$\min_{\alpha,\beta} \left\{ N_\phi(\alpha, \beta) \right\} \geq \frac{2^n}{2} - \tau - (2^n - \phi)$$

Since $\min_{\alpha,\beta} \left\{ N_\phi(\alpha, \beta) \right\} \leq \max_{\alpha,\beta} \left\{ N_\phi(\alpha, \beta) \right\}$, the lower bound in Ramamoorthy's version of H_C can be omitted, and the remaining test is:

$$\max_{\alpha,\beta} \left\{ N_\phi(\alpha, \beta) \right\} \leq \frac{2^n}{2} + \tau$$

This altered definition of H_C is incorporated into the implementations of S-2 given in Section 6.

4.3 Corrected S-7 constraint

The S-7 criteria requires that for each group of input pairs whose difference is the same (and non-zero), at most eight pairs can have the same output difference. In the criteria, this is described as "any nonzero 6-bit difference", which Ramamoorthy et al. appear to have misinterpreted as meaning that all six bits of inputs x and y must differ rather than as a simple description of the length of the difference bit-vector.

A corrected constraint for S-7 for an input XOR i and an output XOR o:

$$\begin{aligned}&\forall i : 1 \leq i < 2^n : \\ &\quad \forall o : 0 \leq o < 2^m : \\ &\qquad 8 \geq |\{\langle x, y\rangle \,|\, 0 \leq x < y < 2^n, x \oplus y = i \wedge S(x) \oplus S(y) = o\}|\end{aligned}$$

Biham and Shamir collect the counts for each difference in a table called the *pairs* XOR *distribution table*. They use ordered pairs instead of unordered pairs as given here; the DES threshold would need to be increased to 16 if applied to their counts [2].

4.4 Reflective symmetry

Ramamoorthy et al. define several types of symmetry: bit-inversion symmetry in which the value of an S-box $S(x)$ is replaced with its inverse, $\neg S(x)$, rotational symmetry for a 180° rotation, row symmetry in which certain rows are exchanged, column symmetry in which certain columns are exchanged, and diagonal symmetry in which blocks of values in the S-box are exchanged diagonally.

We additionally identify a reflective symmetry over the x axis and a reflective symmetry over the y axis.

4.4.1 Reflection over the x-axis

Reflecting an S-box S over the x-axis produces a new S-box, S' in which the values in row 1 are exchanged for those in row 4 and the values in row 2 are exchanged for those in row 3, and vice-versa. Reflecting a 6×4 S-box S, for any input x to S, the value at $S(x)$ will be moved to input $x \oplus 100001$. For a $n \times m$ S-box this is $x \oplus (2^{n-1} + 2^0)$.

For this to be a symmetry, the constraints applied to S should not be violated for S', and conversely. The DES criteria S-1 is already fulfilled by the choice of variables.

Non-linearity criteria S-2

The S-2 criteria requires that no output bit should be "too close" to a linear function of the input bits. This measure of linearity is captured in the count $N(\alpha, \beta)$ and the score σ.

Recall that the linear combination of two bit-vectors x and y of equal length is:

$$\mathsf{linear}(x, y) = (x_0 \wedge y_0) \oplus (x_1 \wedge y_1) \oplus \ldots \oplus (x_{n-1} \wedge y_{n-1})$$

If it is known that a bit k is 0 in x, then $\mathsf{linear}(x + 2^k, y) = \mathsf{linear}(x) \oplus y_k$.

Recall that the count of the S-box is defined as:

$$N(\alpha, \beta) = |\{x \,|\, 0 \leq x < 2^n, \mathsf{linear}(x, \alpha) = \mathsf{linear}(S(x), \beta)\}|$$

It is possible to rewrite $N(\alpha, \beta)$ as a column-based sum (assuming four rows):

$$N(\alpha,\beta) = \sum_{i=0}^{2^{n-2}-1} \begin{pmatrix} 1 \oplus \mathsf{linear}(2i,\alpha) \oplus \mathsf{linear}(S(2i),\beta) + \\ 1 \oplus \mathsf{linear}(2i+1,\alpha) \oplus \mathsf{linear}(S(2i+1),\beta) + \\ 1 \oplus \mathsf{linear}(2i+2^{n-1},\alpha) \oplus \mathsf{linear}(S(2i+2^{n-1}),\beta) + \\ 1 \oplus \mathsf{linear}(2i+2^{n-1}+1,\alpha) \oplus \mathsf{linear}(S(2i+2^{n-1}+1),\beta) \end{pmatrix}$$

The top line of the sum corresponds to the first row of the S-box, the second line to the second row, and so on. Due to the range of i, the 0-th and $n-1$-th bits of $2 \cdot i$ will always be zero and because of this, terms like $\mathsf{linear}(2i + 2^{n-1} + 1)$ can be reduced:

$$N(\alpha,\beta) = \sum_{i=0}^{2^{n-2}-1} \begin{pmatrix} 1 \oplus \mathsf{linear}(2i,\alpha) \oplus \mathsf{linear}(S(2i),\beta) + \\ 1 \oplus \mathsf{linear}(2i,\alpha) \oplus \alpha_0 \oplus \mathsf{linear}(S(2i+1),\beta) + \\ 1 \oplus \mathsf{linear}(2i,\alpha) \oplus \alpha_{n-1} \oplus \mathsf{linear}(S(2i+2^{n-1}),\beta) + \\ 1 \oplus \mathsf{linear}(2i,\alpha) \oplus \alpha_0 \oplus \alpha_{n-1} \oplus \mathsf{linear}(S(2i+2^{n-1}+1),\beta) \end{pmatrix}$$

Reflecting over the x axis alters the values in each column. The corresponding $N'(\alpha, \beta)$ for the reflection of the S-box over the x axis involves swapping the values $S(2i)$ and $S(2i + 2^{n-1} + 1)$ and the values $S(2i + 1)$ and $S(2i + 2^{n-1})$ to get the sum:

$$N'(\alpha,\beta) = \sum_{i=0}^{2^{n-2}-1} \begin{pmatrix} 1 \oplus \mathsf{linear}(2i,\alpha) \oplus \mathsf{linear}(S(2i+2^{n-1}+1),\beta) + \\ 1 \oplus \mathsf{linear}(2i,\alpha) \oplus \alpha_0 \oplus \mathsf{linear}(S(2i+2^{n-1}),\beta) + \\ 1 \oplus \mathsf{linear}(2i,\alpha) \oplus \alpha_{n-1} \oplus \mathsf{linear}(S(2i+1),\beta) + \\ 1 \oplus \mathsf{linear}(2i,\alpha) \oplus \alpha_0 \oplus \alpha_{n-1} \oplus \mathsf{linear}(S(2i),\beta) \end{pmatrix}$$

When a_0 and a_{n-1} are both equal to 0 or both equal to 1, observe that $N(\alpha, \beta) = N'(\alpha, \beta)$. For $a_0 = a_{n-1} = 1$:

$$\begin{aligned} N(\alpha,\beta) &= \sum_{i=0}^{2^{n-2}-1} \begin{pmatrix} 1 \oplus \mathsf{linear}(2i,\alpha) \oplus \mathsf{linear}(S(2i),\beta) + \\ 1 \oplus \mathsf{linear}(2i,\alpha) \oplus 1 \oplus \mathsf{linear}(S(2i+1),\beta) + \\ 1 \oplus \mathsf{linear}(2i,\alpha) \oplus 1 \oplus \mathsf{linear}(S(2i+2^{n-1}),\beta) + \\ 1 \oplus \mathsf{linear}(2i,\alpha) \oplus 1 \oplus 1 \oplus \mathsf{linear}(S(2i+2^{n-1}+1),\beta) \end{pmatrix} \\ &= \sum_{i=0}^{2^{n-2}-1} \begin{pmatrix} 1 \oplus \mathsf{linear}(2i,\alpha) \oplus \mathsf{linear}(S(2i),\beta) + \\ \mathsf{linear}(2i,\alpha) \oplus \mathsf{linear}(S(2i+1),\beta) + \\ \mathsf{linear}(2i,\alpha) \oplus \mathsf{linear}(S(2i+2^{n-1}),\beta) + \\ 1 \oplus \mathsf{linear}(2i,\alpha) \oplus \mathsf{linear}(S(2i+2^{n-1}+1),\beta) \end{pmatrix} \end{aligned}$$

$$N'(\alpha,\beta) = \sum_{i=0}^{2^{n-2}-1} \begin{pmatrix} 1 \oplus \mathsf{linear}(2i,\alpha) \oplus \mathsf{linear}(S(2i+2^{n-1}+1),\beta) + \\ 1 \oplus \mathsf{linear}(2i,\alpha) \oplus 1 \oplus \mathsf{linear}(S(2i+2^{n-1}),\beta) + \\ 1 \oplus \mathsf{linear}(2i,\alpha) \oplus 1 \oplus \mathsf{linear}(S(2i+1),\beta) + \\ 1 \oplus \mathsf{linear}(2i,\alpha) \oplus 1 \oplus 1 \oplus \mathsf{linear}(S(2i),\beta) \end{pmatrix}$$

$$
\begin{aligned}
&= \sum_{i=0}^{2^{n-2}-1} \begin{pmatrix} 1 \oplus \mathsf{linear}(2i,\alpha) \oplus \mathsf{linear}(S(2i+2^{n-1}+1),\beta) + \\ \mathsf{linear}(2i,\alpha) \oplus \mathsf{linear}(S(2i+2^{n-1}),\beta) + \\ \mathsf{linear}(2i,\alpha) \oplus \mathsf{linear}(S(2i+1),\beta) + \\ 1 \oplus \mathsf{linear}(2i,\alpha) \oplus \mathsf{linear}(S(2i),\beta) \end{pmatrix} \\
&= N(\alpha,\beta)
\end{aligned}
$$

The same equality holds if $a_0 = a_{n-1} = 0$.

When $a_0 \neq a_{n-1}$, however, $N'(\alpha,\beta) = 2^n - N(\alpha,\beta)$. For $a_0 = 0, a_{n-1} = 1$:

$$
\begin{aligned}
N(\alpha,\beta) &= \sum_{i=0}^{2^{n-2}-1} \begin{pmatrix} 1 \oplus \mathsf{linear}(2i,\alpha) \oplus \mathsf{linear}(S(2i),\beta) + \\ 1 \oplus \mathsf{linear}(2i,\alpha) \oplus 0 \oplus \mathsf{linear}(S(2i+1),\beta) + \\ 1 \oplus \mathsf{linear}(2i,\alpha) \oplus 1 \oplus \mathsf{linear}(S(2i+2^{n-1}),\beta) + \\ 1 \oplus \mathsf{linear}(2i,\alpha) \oplus 0 \oplus 1 \oplus \mathsf{linear}(S(2i+2^{n-1}+1),\beta) \end{pmatrix} \\
&= \sum_{i=0}^{2^{n-2}-1} \begin{pmatrix} 1 \oplus \mathsf{linear}(2i,\alpha) \oplus \mathsf{linear}(S(2i),\beta) + \\ 1 \oplus \mathsf{linear}(2i,\alpha) \oplus \mathsf{linear}(S(2i+1),\beta) + \\ \mathsf{linear}(2i,\alpha) \oplus \mathsf{linear}(S(2i+2^{n-1}),\beta) + \\ \mathsf{linear}(2i,\alpha) \oplus \mathsf{linear}(S(2i+2^{n-1}+1),\beta) \end{pmatrix} \\
N'(\alpha,\beta) &= \sum_{i=0}^{2^{n-2}-1} \begin{pmatrix} 1 \oplus \mathsf{linear}(2i,\alpha) \oplus \mathsf{linear}(S(2i+2^{n-1}+1),\beta) + \\ 1 \oplus \mathsf{linear}(2i,\alpha) \oplus 0 \oplus \mathsf{linear}(S(2i+2^{n-1}),\beta) + \\ 1 \oplus \mathsf{linear}(2i,\alpha) \oplus 1 \oplus \mathsf{linear}(S(2i+1),\beta) + \\ 1 \oplus \mathsf{linear}(2i,\alpha) \oplus 0 \oplus 1 \oplus \mathsf{linear}(S(2i),\beta) \end{pmatrix} \\
&= \sum_{i=0}^{2^{n-2}-1} \begin{pmatrix} 1 \oplus \mathsf{linear}(2i,\alpha) \oplus \mathsf{linear}(S(2i+2^{n-1}+1),\beta) + \\ 1 \oplus \mathsf{linear}(2i,\alpha) \oplus \mathsf{linear}(S(2i+2^{n-1}),\beta) + \\ \mathsf{linear}(2i,\alpha) \oplus \mathsf{linear}(S(2i+1),\beta) + \\ \mathsf{linear}(2i,\alpha) \oplus \mathsf{linear}(S(2i),\beta) \end{pmatrix}
\end{aligned}
$$

Notice that when the term for a row is added to $N(\alpha,\beta)$, its inverse is added to $N'(\alpha,\beta)$. Since these are Boolean terms and there are four rows and 2^{n-2} columns of terms to be summed, $N(\alpha,\beta) + N'(\alpha,\beta) = 2^{n-2} \cdot 2^2 = 2^n$, which then implies that $N'(\alpha,\beta) = 2^n - N(\alpha,\beta)$. The same equality holds for $\alpha_0 = 1, \alpha_{n-1} = 0$.

Recall that the score, σ, of an S-box is defined as

$$
\sigma = \max_{\alpha,\beta} \left\{ \left| N(\alpha,\beta) - \frac{2^n}{2} \right| \right\}
$$

Let $A = |N(\alpha,\beta) - 2^n/2|$. The corresponding A' for the reflected S-box has two cases, one for $\alpha_0 = \alpha_{n-1}$ and another for $\alpha_0 \neq \alpha_{n-1}$:

$$
\begin{aligned}
A' &= \begin{cases} \left|N'(\alpha,\beta) - 2^n/2\right| & \text{if } \alpha_0 = \alpha_{n-1} \\ \left|N'(\alpha,\beta) - 2^n/2\right| & \text{if } \alpha_0 \neq \alpha_{n-1} \end{cases} \\
&= \begin{cases} |N(\alpha,\beta) - 2^n/2| & \text{if } \alpha_0 = \alpha_{n-1} \\ |2^n - N(\alpha,\beta) - 2^n/2| & \text{if } \alpha_0 \neq \alpha_{n-1} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&= \begin{cases} |N(\alpha,\beta) - 2^n/2| & \text{if } \alpha_0 = \alpha_{n-1} \\ |N(\alpha,\beta) - 2^n/2| & \text{if } \alpha_0 \neq \alpha_{n-1} \end{cases} \\
&= |N(\alpha,\beta) - 2^n/2| \\
&= A
\end{aligned}
$$

So the score for the reflected S-box, σ', is the same as σ

$$\sigma' = \max_{\alpha,\beta} \{A'\} = \max_{\alpha,\beta} \{A\} = \sigma$$

Since the reflected S-box has the same score as the original S-box, the original S-box fulfills criteria S-2 if and only if the reflected S-box also fulfills criteria S-2.

Criteria S-3

Criteria S-3 requires that the values of each row are distinct; the original S-box has this property if and only if reflected S-box also has this property.

Criteria S-4

Criteria S-4 is a constraint on pairs of variables, x and y, whose inputs differ by a single bit, that is that $\mathsf{weight}(x \oplus y) = 1$. In the reflected S-box, S', the input x is moved to $x \oplus (2^{n-1} + 2^0)$ and the input y is moved to $y \oplus (2^{n-1} + 2^0)$. Because $x \oplus (2^{n-1} + 2^0) \oplus y \oplus (2^{n-1} + 2^0) = x \oplus y$, reflecting S to S' maintains this relationship between x and y, so criteria S-4 is fulfilled in S if and only if it is also fulfilled in S'.

Criteria S-5

Criteria S-5 is a constraint on pairs of variables whose inputs differ in the "two middle bits exactly." Variables which have this property belong to the same row of the S-box, and these middle bits are not affected by the reflection, so this constraint is fulfilled by S if and only if it is also fulfilled by S'.

Criteria S-6

Criteria S-6 is a constraint on pairs of variables whose inputs differ in their first two bits, but are the same in their last two.

For a given input x, the inputs related to x are variables y such that $(x \oplus y) \wedge (2^{n-1} + 2^{n-2} + 2^1 + 2^0) = (2^{n-1} + 2^{n-2})$. In the reflected S-box, S', the input x is moved to $x \oplus (2^{n-1} + 2^0)$ and y is moved to $y \oplus (2^{n-1} + 2^0)$. And because $x \oplus (2^{n-1} + 2^0) \oplus y \oplus (2^{n-1} + 2^0) = (x \oplus y)$, the relationship between these inputs is maintained in S'.

Criteria S-7

Criteria S-7 places a constraint on sets of pairs of variables whose inputs x and y differ for some input difference i: $x \oplus y = i$.

Again, because $x \oplus (2^{n-1} + 2^0) \oplus y \oplus (2^{n-1} + 2^0) = (x \oplus y)$, this relationship is unchanged for the reflected S-box S', so criteria S-7 is fulfilled in S if and only if it is also fulfilled in S'.

4.4.2 Reflection over the y-axis

We have shown that reflection over the x axis is a symmetry, while Ramamoorthy et al. shown a 180° degree rotational symmetry. A reflection over the y-axis is equivalent to a reflection over the x axis followed by a 180° rotation.

If an S-box S fulfills criteria S-1–S-7, then so does an S-box S' which is reflected over the x-axis, and so does an S-box S'' which is S' rotated by 180°, which in turn is equivalent to an S-box reflected over the y-axis.

4.4.3 Symmetry-breaking

During search, reflective symmetry over the x-axis can be broken by constraining the variable at the top-left of the S-box to be less than or equal to the variable at the bottom-left of the S-box. For a 6×4 S-box, these are the variables for input 0 and 33: $I(S(0)) \leq I(S(33))$

The reflective symmetry over the y-axis can be broken during search by constraining the variable at the top-left of the S-box to be less than or equal to the variable at the top-right of the S-box. For a 6×4 S-box, these are the variables for input 0 and 30: $I(S(0)) \leq I(S(30))$

Chapter 5

Bit-vector implementation

The implementation of variables for Gecode consists of defining *variable implementations*, *variables* and *variable arrays*, *variable views*, *propagators*, and *branchings*. This process is covered extensively by *Modelling and Programming with Gecode*, section V: *Programming Variables* [14].

Variable implementations are the objects which actually store the variable domains and which implement the operations which manipulate the domain. Variables are read-only interfaces to variable implementations that enable multiple variables to refer to the same implementation. Variable arrays are just arrays of variables. Variable views allow Gecode to decouple variable implementations from propagator implementations. Propagators are the implementations of constraints for a given variable type. Branchings consist of variable and value selection functions which choose variables on which to branch and which value to fix for each branch of the search tree.

Presented below are the implementation details for each of these components for the bit-vector variables and propagators implemented by this thesis.

5.1 Variable implementations

The domain of the bit-vector variable implementation is defined by a pair of bit-vectors, $\langle l, u \rangle$, as described in Section 3.1. These upper and lower bounds of the bit-vector variables are represented as `unsigned int`s (which is typically the same as a machine's word size) [10]. The variable implementation also contains an `unsigned int` n, the number of bits to use for the variable, and a bit-mask m which masks the lower n bits.

A new variable type must be compiled into Gecode by way of a *variable implementation specification* file which defines the variable name, its C++ namespace, the possible modification events, and the possible propagation conditions.

Modification events describe how the domain of a view has changed. All variable implementations must provide modification events for `NONE`, `FAILED`, and `ASSIGNED`. We define these modification events and the additional events for

modification event	description
ME_BIT_NONE	no change to the variable domain
ME_BIT_FAILED	the variable domain is failed
ME_BIT_VAL	the variable has become assigned
ME_BIT_LOWER	the lower bound of the variable domain has changed
ME_BIT_UPPER	the upper bound of the variable domain has changed
ME_BIT_BND	both the upper and lower bounds have changed

Table 5.1: Bit-vector modification events

when the the lower and upper bounds of a variable's domain change (`LOWER` and `UPPER`) or when both change (`BND`). All modification events are listed in Table 5.1.

Propagation conditions describe when a propagator is scheduled depending on how the views to which it is subscribed are modified. All variable implementations must provide propagation conditions for `NONE` and `ASSIGNED` (aliased to `VAL`). We additionally define propagation conditions for when a view changes its upper, lower, or both bounds (`UPPER`, `LOWER`, `BND`), summarized in Table 5.2.

Variable implementations must implement an `assigned` function, which returns `true` when the implementation becomes assigned; in the case of bit-vectors, this is when `l == u`.

The implementation must also provide `subscribe` and `cancel` functions which are used by propagators to be notified upon certain propagation conditions, functions for copying the variable during search, and access operations to the relevant member variables, for bit-vectors these are `lower` and `upper` for the variable bounds, `num_bits` for the number of bits n and `mask` for the mask of the lower n bits.

Finally, the implementation must provide some way of modifying the variable domain and ensure that these modification operations return the correct modification events. For bit-vectors, these are `lower` and `upper` which set the lower or upper bound to a passed value, and `bounds`, which sets both the lower and upper bounds simultaneously.

Gecode supports a construct called an advisor which is used to improve incremental propagation; these in turn use *deltas* to communicate changes in variable domains. The bit-vector variable implementation does not support this feature and therefore does not provide deltas to be used by advisors; it indicates this by the `GECODE_NEVER` macro in the relevant functions.

5.2 Variables & variable arrays

Variables are read-only interfaces to variable implementations, where each implementation can be referred to by an arbitrary number of variables [14]. The code to be written for a variable is minimal and consists of a few constructors and a pair of access operations which pass directly through to the variable implementation, `lower` and `upper`.

propagation condition	description
PC_BIT_NONE	the propagator will not create any subscriptions
PC_BIT_VAL	will schedule a propagator when a subscribed variable is assigned
PC_BIT_LOWER	will schedule a propagator when a subscribed variable's lower bound is altered
PC_BIT_UPPER	will schedule a propagator when a subscribed variable's upper bound is altered
PC_BIT_BND	will schedule a propagator when a subscribed variable's upper or lower bound is altered

Table 5.2: Bit-vector propagation conditions

The code for variable arrays is similarly simple; the constructors simply initialize the relevant number of variables.

5.3 Variable views

Variable views are used by Gecode to decouple the variable implementations from propagator implementation. Each view implements the same operations as a variable. Gecode supports *variable implementation views* which maintain a reference to a variable and directly invoke operations on the backing variable, *constant views* where the view behaves as a variable equal to some constant c without the need for a backing variable, and *derived views* which reference another view instead of a variable [16]. Propagators can then be implemented generically to operate on these views [15].

Modelling and programming with Gecode gives examples of some possible derived views for integer variables: an *offset view*, which adds a constant factor c and a *scale view*, which scales the variable value by a constant a [14].

We implement a variable implementation view and a constant view for bit-vectors.

5.4 Propagators

The code necessary to implement the `propagate` function in Gecode can be very similar to the pseudocode defined in Sections 3.1 and 4.1. The `propagate` function for the XOR propagator implementation is given in Listing 5.1 and an altered version of the XOR propagator which corresponds closely to the Gecode implementation is given in Algorithm 5.1. For example, lines 11–12 in Listing 5.1 correspond to lines 2–3 in Algorithm 5.1, while line 13 corresponds with lines 4–5.

Listing 5.1 XOR propagate function

```
ExecStatus
Exor::propagate(Space& home,
                const ModEventDelta&) {
    BitType ux = x0.upper();
    BitType uy = x1.upper();
    BitType uz = x2.upper();
    BitType lx = x0.lower();
    BitType ly = x1.lower();
    BitType lz = x2.lower();

    BitType lowx = lx | ((~ uz) & ly) | (lz & (~ uy));
    BitType upx  = ux & (uz | uy) & (~ (ly & lz));
    GECODE_ME_CHECK(x0.bounds(home, upx, lowx));

    BitType lowy = ly | ((~ uz) & lx) | (lz & (~ ux));
    BitType upy  = uy & (uz | ux) & (~ (lx & lz));
    GECODE_ME_CHECK(x1.bounds(home, upy, lowy));

    BitType lowz = lz | ((~ ux) & ly) | (lx & (~ uy));
    BitType upz  = uz & (ux | uy) & (~ (lx & ly));
    GECODE_ME_CHECK(x2.bounds(home, upz, lowz));

    if(x0.assigned() && x1.assigned() && x2.assigned()) {
        return home.ES_SUBSUMED(*this);
    }
    return ES_FIX;
}
```

Algorithm 5.1 Altered propagator for $x \oplus y = z$

function propagate($x \oplus y = z$)
  $low^x \leftarrow l^x \vee (\neg u^z \wedge l^y) \vee (l^z \wedge \neg u^y)$
  $up^x \leftarrow u^x \wedge (u^z \vee u^y) \wedge \neg(l^y \wedge l^z)$
  $\langle l^x, u^x \rangle \leftarrow \langle low^x, up^x \rangle$
  if $\neg$ valid$(\langle l^x, u^x \rangle)$ **then**
    return false
  $low^y \leftarrow l^y \vee (\neg u^z \wedge l^x) \vee (l^z \wedge \neg u^x)$
  $up^y \leftarrow u^y \wedge (u^z \vee u^x) \wedge \neg(l^x \wedge l^z)$
  $\langle l^y, u^y \rangle \leftarrow \langle low^y, up^y \rangle$
  if $\neg$ valid$(\langle l^y, u^y \rangle)$ **then**
    return false
  $low^z \leftarrow l^z \vee (\neg u^x \wedge l^y) \vee (l^x \wedge \neg u^y)$
  $up^z \leftarrow u^z \wedge (u^x \vee u^y) \wedge \neg(l^x \wedge l^y)$
  $\langle l^z, u^z \rangle \leftarrow \langle low^z, up^z \rangle$
  if $\neg$ valid$(\langle l^z, u^z \rangle)$ **then**
    return false
  return true
end function

5.5 Branchings

The implementation of branchings in Gecode requires *variable selection* and *value selection* functions, a default *merit* function, and a *branch commit* function [14]. The bit-vector implementation provides a number of variable-value branchings.

Variable selection functions operate on a number of decision variables and choose best suited variable on which to branch based on the given selection strategy. For bit-vector variables, the possible variable selection functions are given in Table 5.3. The variable selection code itself is mostly boilerplate, which uses Gecode `ViewSel` classes to do the actual variable selection.

The variable selection functions involving the merit make use of a merit function. By default, the bit-vector merit function returns the number of free bits. This function can be overridden by a user-defined function, which allows for custom variable selection.

variable selection	**select...**
`BIT_VAR_NONE`	the first unassigned variable
`BIT_VAR_RND`	a random variable
`BIT_VAR_SIZE_MIN`	the variable with the smallest domain size
`BIT_VAR_MERIT_MAX`	the variable with highest merit as defined by `merit`
`BIT_VAR_DEGREE_MAX`	the variable on which most propagators depend
`BIT_VAR_ACTIVITY_MAX`	the variable which has been involved in most constraint propagation

Table 5.3: Bit-vector branching variable selection

value selection	**select...**
`BIT_VAL_MIN_BIT`	the least-significant free bit
`BIT_VAL_MAX_BIT`	the most significant free bit
`BIT_VAL_RND_BIT`	a random free bit
`BIT_VAL`	user-defined value selection

Table 5.4: Bit-vector branching value selection

Value selection functions operate on the selected variable and choose a value in the domain of this variable on which to branch. In the case of bit-vectors, the function chooses a single bit according to the chosen strategy. The defined value selection functions are given in Table 5.4. The value selection functions require *find first set bit* and *find last set bit* functions in order to find a bit to select. Our implementation uses the `gcc` built-in functions `__builtin_ffs` (find first set) and `__builtin_clz` (count leading zeros), which internally use the `bsfl` (bit scan forward) and `bsrl` (bit scan reverse) instructions [7]. These run in constant time, which allows our selection functions to run in constant time.

Finally, branch commit function branches on the chosen variable and value. For bit-vectors, the default branch commit function first fixes the selected bit to 1, then fixes the selected bit to 0.

Chapter 6

Bit-vector S-Box models

In this chapter, we present several model variants for enforcing the DES design criteria for S-boxes using bit-vector variables. Later, in Chapter 7, two alternative models are presented; one using set variables and another using Boolean variables.

6.1 Variable choice

All bit-vector-based models represent the problem as an array of 2^n bit-vector variables which have m "active" bits. Criteria S-1 is fulfilled by this choice of variables.

6.2 Channeling

All bit-vector models channel their variables to an integer variable representation in order to take advantage of the built-in Gecode implementation of alldifferent and in some cases count. The channeled representation is an array of 2^n integer variables.

The channeling constraint for bit-vectors is as presented in Algorithm 3.5. For some bit-vector models, the integer representations are used in S-2, the non-linearity constraint. All bit-vector models use integer representations in S-3.

6.3 Criteria S-2

Criteria S-2, the non-linearity constraint, is enforced by some variant of the H_C heuristic presented in Section 3.2.1. This constraint can either be expressed as a global constraint or series of more basic constraints.

6.3.1 "Decomposed" bit-vector S-2

The H_C heuristic can be expressed as a set of constraints as presented in Algorithm 6.1.

For each combination of α and β and every S-box input x, an entry in the array *Nab* of Boolean variables is constrained to be true when the linear combination of x and α equals the linear combination of $S(x)$ and β. An integer variable in the array N is then constrained to be the sum of *Nab* for the given α and output bit i. Finally, the max and min of N are constrained to be within the acceptable range given the threshold τ.

Algorithm 6.1 Model for the S-2 non-linearity constraint

for $1 \leq \alpha < 2^n$ **do**
  for $0 \leq i < m$ **do**
    $\beta \leftarrow 2^i$
    for $0 \leq x < 2^n$ **do**
      $Nab[\alpha][i][x] = (\mathsf{parity}(x \wedge \alpha) \Leftrightarrow \mathsf{parity}(S(x) \wedge \beta))$
    $N[\alpha - 1][i] = \mathsf{sum}(Nab[\alpha][i])$
$2^n/2 + \tau \geq \mathsf{max}(N)$
$2^n/2 - \tau \leq \mathsf{min}(N)$

6.3.2 Global integer & bit-vector S-2

A global propagator for the S-2 non-linearity constraint is given in Algorithm 6.2. We implement two variants of nonlinear; one which operates on integer variables and one which operates directly on bit-vector variables.

For all assigned variables, and for all potential combinations of α and β, the propagator increments $N(\alpha, \beta)$ when the linear combination of the current input, x, and α is equal to the linear combination of the current variable, $S(x)$, and β. The propagator keeps a count of the number of assigned variables, *assigned_count*, and calculates the max and min of $N(\alpha, \beta)$. If the maximum or minimum values are not within the acceptable range, given the threshold τ and *assigned_count*, then the propagator reports failure.

The advantage of using a global propagator in this case is a much reduced number of variables to which changes must be propagated. The implementation of this propagator additionally keeps track of which variables have been assigned and incrementally updates $N(\alpha, \beta)$ in order to improve execution time, rather than completely recalculating $N(\alpha, \beta)$ on each run.

Algorithm 6.2 Global propagator for nonlinear(S, τ)

```
function propagate(nonlinear(S, τ))
    for 1 ≤ α < 2^n do
        for 0 ≤ i < m do
            β ← 2^i
            N(α, β) ← 0
    assigned_count ← 0
    for 0 ≤ x < 2^n do
        if assigned(S(x)) then
            assigned_count ← assigned_count + 1
            for 1 ≤ α < 2^n do
                for 0 ≤ i < m do
                    β ← 2^i
                    if parity(x ∧ α) = parity(S(x) ∧ β) then
                        N(α, β) ← N(α, β) + 1
    max ← max_{α,β} {N(α, β)}
    min ← min_{α,β} {N(α, β)}
    if ¬(2^n/2 + τ ≥ max ∧ min ≥ 2^n/2 − τ − (2^n − assigned_count)) then
        return false
    return true
end function
```

6.4 Criteria S-3

Criteria S-3 requires that each possible value 0–$(2^m - 1)$ occur at least once; this is enforced by an alldifferent constraint on the channelled integer variables as described in Section 3.2.2.

6.5 Criteria S-4, S-5, and S-6

Criteria S-4, S-5, and S-6 are implemented as described in sections 3.2.3, 3.2.4, and 3.2.5. These use the bit-vector propagators for weight, xor and disequal as described in Section 3.1 and Section 4.1.

6.6 Criteria S-7

Criteria S-7, described in Section 4.3, is expressed either as a global constraint or as a "decomposed" series of basic constraints. The criteria specifies *max_count* to be 8 for a 6×4 S-box.

6.6.1 "Decomposed" bit-vector S-7

The S-7 criteria can be expressed as a series of constraints as presented in Algorithm 6.3. For each input difference, i, each pair of inputs j and k are found which result in this difference. Because we are using a *max_count* for

unordered pairs, we ensure that $j < k$ to avoid duplicates. The output difference $S(j) \oplus S(k)$ is channeled to an integer representation. Once all input pairs for this input difference are found, a global cardinality constraint (built-in to Gecode as count) constrains the variable at index *idx* of the integer variable array *counts* to be equal to the number of occurrences of the value *idx* in $xorintvals[i]$. Finally, these counts are constrained to be less than or equal to *max_count*.

Algorithm 6.3 Model for the S-7 constraint

for $1 \leq i < 2^n$ **do**
  $cur \leftarrow 0$
  for $0 \leq j < 2^n$ **do**
    $k \leftarrow i \oplus j$
    if $j < k$ **then**
      $xorintvals[i][cur] = \mathsf{channel}(S(j) \oplus S(k))$
      $cur \leftarrow cur + 1$
  $counts = \mathsf{count}(xorintvals[i])$
  $counts \leq max\_count$

6.6.2 Global bit-vector S-7

We also implement a global propagator for criteria S-7, as given in Algorithm 6.4, called xordist in reference to the pairs XOR distribution table mentioned in Section 4.3. The propagator operates on a slightly different principle than the model presented in the previous section. Instead of starting with an input difference i, it iterates over all inputs x, and all inputs y such that $x < y$, making ordered pairs of inputs $\langle x, y \rangle$ with the input difference $x \oplus y$. The count in the table T for the output difference for these inputs $S(x) \oplus S(y)$ is then incremented. If at any point the count exceeds the *max_count*, the propagator reports failure.

Like the global nonlinear constraint presented in 6.3.2, the global xordist reduces the number of variables to which changes must be propagated to achieve the same constraint, and like nonlinear, xordist also incrementally updates T in order to improve execution time.

Algorithm 6.4 Global propagator for criteria S-7

function propagate(xordist(S, max_count))
  for $0 \leq x < 2^n$ **do**
    if assigned($S(x)$) **then**
      for $x < y < 2^n$ **do**
        if assigned($S(y)$) **then**
          $T[x \oplus y][S(x) \oplus S(y)] \leftarrow T[x \oplus y][S(x) \oplus S(y)] + 1$
          if $T[x \oplus y][S(x) \oplus S(y)] > max\_count$ **then**
            return false
  return true
end function

6.7 Models

Four bit-vector model variants are constructed:

- A bit-vector model using the "decomposed" S-2 and S-7 constraints
- A bit-vector model using the global integer S-2 propagator and the "decomposed" S-7 constraint
- A bit-vector model using the global bit-vector S-2 propagator and the "decomposed" S-7 constraint
- A bit-vector model using both the global bit-vector S-2 propagator and the global S-7 propagator

Chapter 7

Alternative S-Box models

This chapter details two alternative S-box models using set variables and Boolean variables to represent the S-box's constituent bit-vectors.

7.1 Set model

The set-based model represents an S-box as an array of 2^n set variables, each of which can contain values from 0 to m; if a value k is present in a set, it indicates that bit k is set to 1, otherwise the bit is set to 0.

7.1.1 Channeling

Set variables are channeled to integer representation. The channeling is achieved by using the Gecode-provided weights constraint, which gives a weight to each possible set element and constrains an integer variable to be the sum of the weights for the elements present in the set.

7.1.2 Bitwise operations

For the set model, the built-in set operations described in *Modelling and Programming with Gecode* [17] are used to implement "bitwise" operations on sets representing "on" bits. The bitwise XOR ($\oplus$) operation over two variables p and q, $p \oplus q$, is equivalent to $(p \vee q) \wedge \neg(p \wedge q)$. For the representation described here, the set union ($\cup$) is equivalent to $\vee$, the set intersection ($\cap$) is equivalent to $\wedge$, and the set complement (S^C for a set S) is equivalent to $\neg$. Thus, bitwise $\oplus$ is implemented for two sets p and q as $(p \cup q) \cap (p \cap q)^C$. Set cardinality is here equivalent to the hamming weight described in Section 2.2.6.

7.1.3 Criteria S-2

The non-linearity criteria S-2 uses the channeled integer representations and the global integer S-2 described in Section 6.3.2.

7.1.4 Criteria S-3

This criteria is implemented in exactly the same way as for the bit-vector models, as described in Section 6.4.

7.1.5 Criteria S-4, S-5, and S-6

As with S-3, the set model variants of S-4, S-5, and S-6 correspond with the bit-vector models. The set model uses the set-based interpretation of bitwise operations described in section 7.1.2.

7.1.6 Criteria S-7

The set-based implementation of criteria S-7 corresponds with the "decomposed" bit-vector S-7 implementation presented in Section 6.6.1, but with the set-based interpretation of bitwise operations described in section 7.1.2.

7.1.7 Comparison

The set-based model builds on existing variable implementations, which have the potential to be more thoroughly optimized than our bit-vector variable implementation. However, this set-based model also requires some acrobatics to express bitwise operations which can result in a significant number of auxiliary variables.

7.2 Boolean model

The Boolean-based model represents an S-box as an array of 2^n arrays of 2^m Boolean variables, where each array represents a bit-vector and variable in the array represents a single bit.

7.2.1 Channeling

Each Boolean array representing a bit-vector is channeled to a single integer variable using the built-in Gecode integer linear constraint. For a Boolean array, the linear constraint interprets each true value as 1 and each false value as 0 and constrains an integer variable to be the linear combination of the array and a set of weights. The weights in this case are 2^k for each possible bit k, $0 \leq k < m$. Note that this is not the same as the bit-vector linear operation described in Section 2.2.5.

7.2.2 Bitwise operations

Bitwise operations are more easily represented with arrays of Boolean variables than with sets; for each index i in a pair of Boolean arrays, p and q, the Boolean operation can be directly applied to the corresponding indices. For example, performing $o = p \oplus q$ is just $\forall i : 0 \leq i < m : o[i] \Leftrightarrow p[i] \oplus q[i]$.

7.2.3 Criteria S-2

For the Boolean model, the non-linearity criteria S-2 uses the channeled integer representations and the global integer S-2 described in Section 6.3.2.

7.2.4 Criteria S-3

For the Boolean model, criteria S-3 is implemented in exactly the same way as for the bit-vector models, as described in Section 6.4.

7.2.5 Criteria S-4, S-5, and S-6

As with S-3, the Boolean model variants of S-4, S-5, and S-7 correspond with the bit-vector models. The Boolean model uses the Boolean bitwise operations described in Section 7.2.2.

7.2.6 Criteria S-7

The Boolean implementation of criteria S-7 corresponds with the "decomposed" bit-vector S-7 implementation presented in Section 6.6.1, but with the Boolean bitwise operations described in section 7.2.2 and channeling to integer variables as described in Section 7.2.1.

7.2.7 Comparison

Like the set-based model, the Boolean model builds on existing variable implementations, which may be better optimized than our bit-vector variable implementation. The Boolean model requires more variables than the set-based model, but uses fewer interim steps to perform the bitwise operations. Internally, Gecode uses integer variables to represent Boolean variables, which will not be as memory-efficient when compared with bit-vector variables.

Chapter 8

Evaluation

This chapter describes the set-up and results of benchmarking run on the set, Boolean and four different bit-vector models.

8.1 Setup

In order to ensure that a solution exists, one of the DES-defined S-boxes was chosen as a basis for the search space.

For all models, we use all symmetry breaking constraints as defined in Sections 3.3 and 4.4. However, no S-box defined by DES fulfills all of the defined symmetries, so we transform the chosen S-box into one which fulfills these symmetries by inverting bits, transposing rows, rotating the S-box, etc., as appropriate.

A set of randomly-chosen values are then removed from the S-box and the time necessary to find the specific S-box solution is measured for 25 runs. These values are selected randomly for each run but the same values are used for for all model implementations.

Four variable selection methods were evaluated: `NONE` (choose first unassigned variable), `DEGREE_MAX` (choose the variable involved in most constraints), `ACTIVITY_MAX` (choose the variable with most activity), and `RND` (choose a random variable).

For the bit-vector and set models, the variable selection targets an entire bit-vector. In the Boolean model, in contrast, the variable selection targets individual bits.

For bit-vector models, value selection was `BIT_VAL_RND_BIT`, which first fixes a randomly-chosen bit to 1, then to 0. Value selection for the set-based model was `SET_VAL_RND_INC`, which includes a random element in the set; this is equivalent to setting a random bit to 1. For the Boolean-based model, the value selection was `INT_VAL_MAX()`, which first fixes the chosen bit to 1, then to 0.

Boolean, integer S-2

	ACTIVITY			DEGREE			NONE			RND		
n	time	t/o	nodes	time	t/o	nodes	time	t/o	nodes	time	t/o	nodes
8	0.02		21	0.02		21	0.02		21	0.02		21
12	0.05		56	0.04		43	0.04		43	0.11		155
16	0.12		154	0.10		112	0.10		112	0.83		1228
20	0.64		769	0.19		218	0.20		218	14.79		18972
24	5.01	(1)	5950	0.56		674	0.58		674			
28				1.99		2330	1.91		2330			
32				4.51		6441	4.75		6441			

set, integer S-2

	ACTIVITY			DEGREE			NONE			RND		
n	time	t/o	nodes	time	t/o	nodes	time	t/o	nodes	time	t/o	nodes
8	0.05		29	0.14		87	0.05		28	0.09		55
12	0.11		59	1.06		635	0.10		54	0.60		353
16	0.27		157	10.35	(2)	5467	0.20		110	4.92		2695
20	1.01		581				0.46		245			
24	3.45	(2)	1764				2.15		1165			
28							6.20	(1)	3687			

bit-vector, decomposed S-2

	ACTIVITY			DEGREE			NONE			RND		
n	time	t/o	nodes	time	t/o	nodes	time	t/o	nodes	time	t/o	nodes
8	0.03		18	0.10		69	0.04		17	0.06		39
12	0.08		41	1.38		782	0.07		37	0.35		214
16	0.18		88				0.18		80	3.25		1798
20	0.66		315				0.44		204			
24	2.15		936				2.81		1115			
28							5.95		2853			

Table 8.1: Average execution time in seconds, total number of timeouts, average nodes over 25 runs. Boldface values are the best time for the given number of values to find, n.

Each combination of model, variable selection, and n, the number of elements to remove from the chosen S-box, $n \in \{4, 8, 12, \ldots, 64\}$, is tested for 25 runs. The threshold for the nonlinearity criteria is set to 14 to match the score of S_3, the selected S-box.

A timeout of 30 seconds was used. If a model timed out less than 3 times, we report the number of timeouts and its average time and nodes for the remaining runs. Once a model times out 3 times for a given n, we abort execution for the remaining n.

Tests were performed under Debian Linux on a virtual private server with an emulated CPU (AuthenticAMD, QEMU Virtual CPU version 0.12.5, 1909.490 MHz, 512 KB cache) and 500 MB of RAM. The DES S-boxes are 6×4 substitution boxes, meaning they consist of 64 4-bit variables.

bit-vector, integer S-2

	ACTIVITY			DEGREE			NONE			RND		
n	time	t/o	nodes	time	t/o	nodes	time	t/o	nodes	time	t/o	nodes
8	0.02		18	0.04		70	0.02		18	0.03		40
12	0.03		41	0.45		786	0.03		37	0.12		215
16	0.06		88	12.42		21898	0.06		80	0.98		1782
20	0.20		320				0.14		204	10.81		19630
24	0.60		937				0.71		1117			
28	4.57	(1)	7834				1.37		2860			
32							3.95		8978			

bit-vector, S-2

	ACTIVITY			DEGREE			NONE			RND		
n	time	t/o	nodes	time	t/o	nodes	time	t/o	nodes	time	t/o	nodes
8	0.02		18	0.04		71	0.03		18	0.03		40
12	0.03		41	0.44		786	0.05		37	0.12		212
16	0.06		88	12.35		21898	0.10		80	0.98		1782
20	0.21		320				0.19		204	10.82		19630
24	0.60		937				1.13		1117			
28	4.58	(1)	7834				1.94		2860			
32							5.02	(1)	5810			

bit-vector, S-2, global S-7

	ACTIVITY			DEGREE			NONE			RND		
n	time	t/o	nodes	time	t/o	nodes	time	t/o	nodes	time	t/o	nodes
8	**0.01**		18	**0.01**		71	**0.01**		18	**0.01**		40
12	**0.01**		41	0.05		786	**0.01**		37	0.02		211
16	**0.01**		88	1.24		21569	**0.01**		80	0.10		1683
20	0.03		333				**0.02**		205	1.16		20926
24	0.10		1542				**0.08**		1119			
28	0.63		12180				**0.14**		2862			
32	2.87	(1)	49179				**0.44**		8989			
36							**2.01**		41245			
40							**6.66**	(1)	130116			

8.2 Results

The average execution time, number of timeouts, and average number of nodes are presented in Table 8.1. The best times for a given n are shown in bold.

The RND value selection performed poorly for all models. Figure 8.1 graphs the results of each model for RND value selection and shows the relative advantage of the bit-vector model with the bit-vector S-2 and S-7 propagators. The bit-vector model using the decomposed S-2 and S-7 constraints, performed the worst, though only slightly worse than the set-based model. Presumably, RND performed poorly because many of the propagators require variables to become assigned before they execute and selecting random variables delays this process.

The DEGREE value selection also performed poorly. The results for each model and DEGREE value selection are graphed in Figure 8.2, where it can be seen that the only successful case is the Boolean model. This is likely down to the fact that the value selection for the Boolean model is targeting individual bits rather than entire bit-vectors, unlike the other models. Each bit-vector is involved in the same number of constraints, while individual bits may be involved in more or less.

ACTIVITY value selection improved on the previous two value selection methods. Figure 8.3 shows the Boolean model performing the worst, followed by the set model, then the bit-vector models, with the model using bit-vector S-2 and global S-7 propagators performing the best. The poor performance of the Boolean model in this case is again likely due to the fact that value selection targets individual Boolean variables representing bits, which are either assigned or not and therefore cannot accumulate much activity.

Finally, the best performing value selection was NONE. The best performing model was the bit-vector model using the bit-vector S-2 and global S-7 propagators, while the worst performing model was the bit-vector model with the decomposed S-2 and S-7 constraints, as shown in Figure 8.4. This value selection probably performed well for the same reasons that RND performed poorly, namely that it causes variables to become assigned more quickly and therefore increases propagation.

Overall, the bit-vector model with global bit-vector propagators for both S-2 and S-7, in combination with the NONE value selection performed the best for all values of n; no other model/value-selection combination reached $n = 40$.

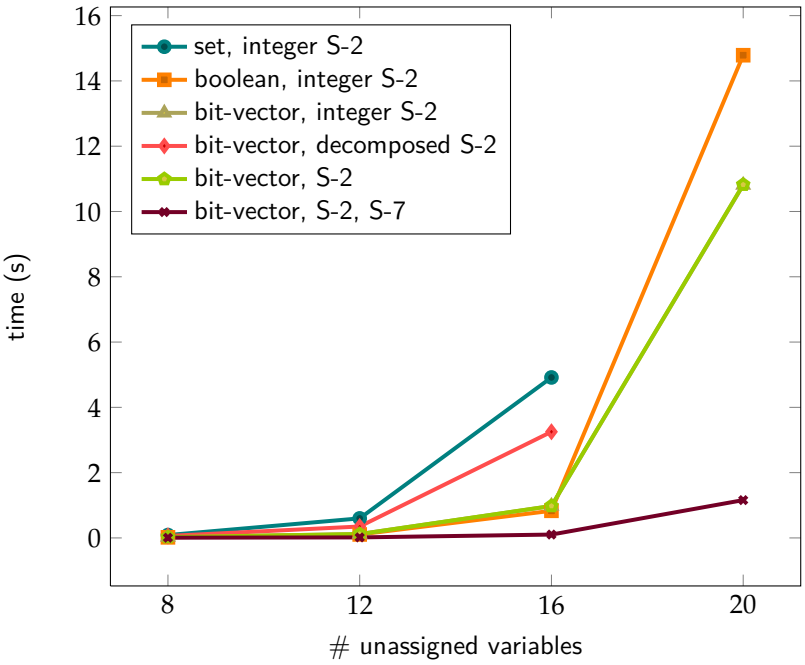


Figure 8.1: RND

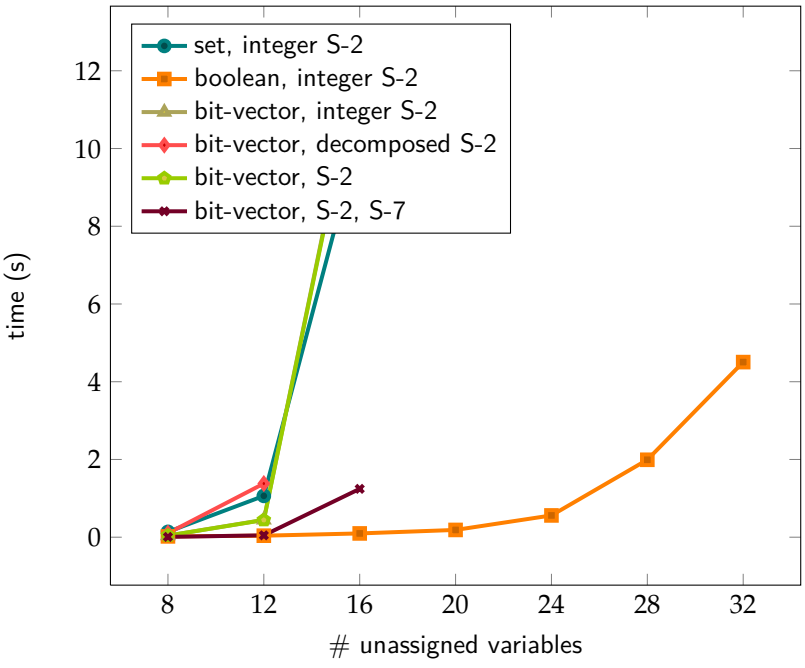


Figure 8.2: DEGREE

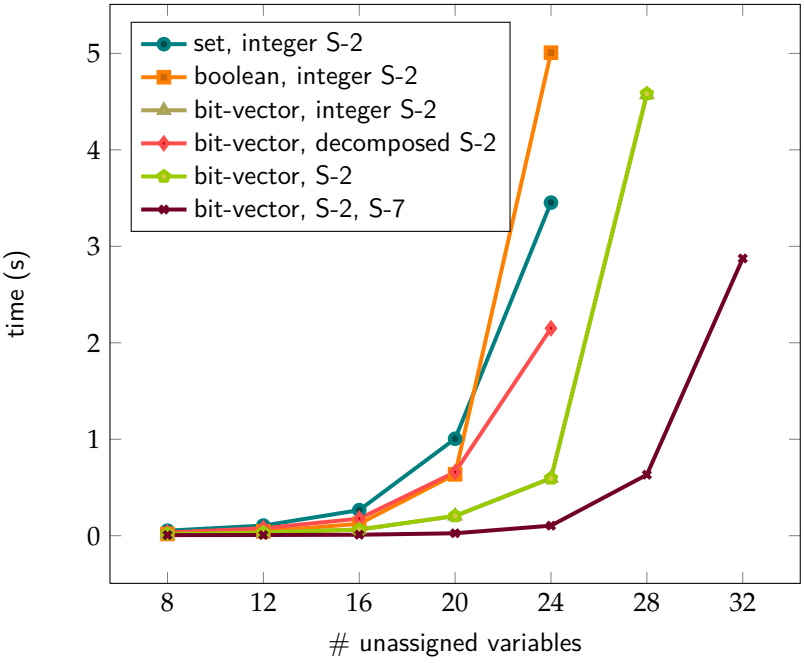


Figure 8.3: ACTIVITY

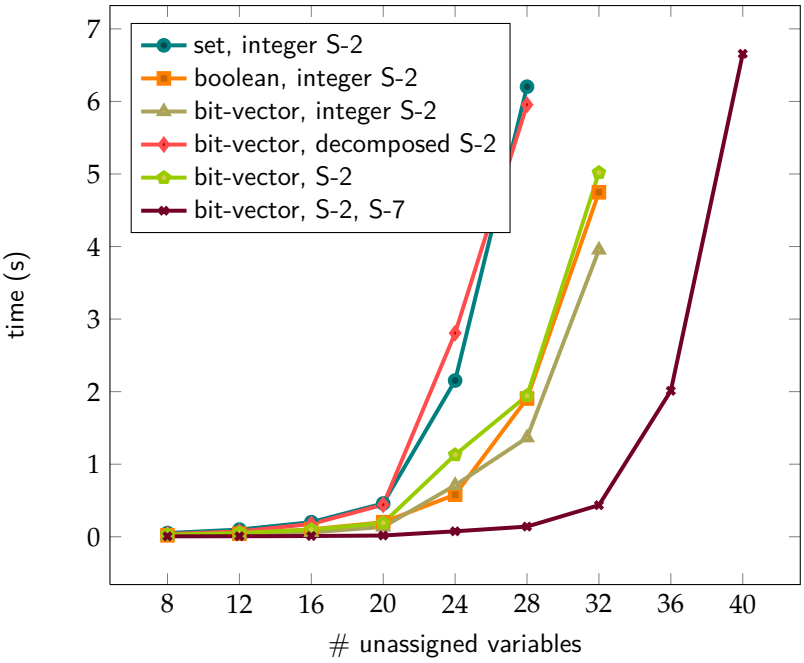


Figure 8.4: NONE

Chapter 9

Conclusion

We have presented a bit-vector variable implementation for Gecode and its application to the problem of finding high-quality cryptographic substitution boxes.

We have described constraint programming, bit-vectors, and substitution boxes, and reviewed previous work on bit-vectors in constraint programming and the application of constraint programming to substitution box generation.

We have corrected errors in [12], presented two additional symmetries for substitution boxes, and defined several generic bit-vector propagators and global propagators for the S-2 and S-7 requirements.

The bit-vector variable implementation was used in several models for substitution box generation, and tested against models using set and Boolean variables for different variable selection methods. Our testing indicates that the best combination of model and variable selection was a model based on bit-vector variables using global propagators for the S-2 and S-7 constraints in conjunction with the `NONE` variable selection.

9.1 Future work

We implemented only a subset of the bit-vector propagators defined by Michel and Van Hentenryck, and they define propagators for only a subset of logical operations. Future work could implement the defined propagators for Gecode and provide definitions for other logical operations. Furthermore, there are additional operations on bit-vectors which may be of interest; for example the find first or find last set bit operations.

We implemented only the DES-defined S-2 constraint; a trivial to implement extension would be to use Matsui's S-2′ instead. This work is based heavily on the DES design criteria, but there may be additional constraints which would provide better substitution boxes.

As defined, the global propagators for requirements S-2 and S-7 only fail a space and provide no propagation; it may be possible to improve these require-

ments in such a way that the propagators can actually perform propagation, which in turn could lead to better runtimes.

Our models use the integer-based alldifferent constraint, and place constraints on the channeled integer representations of bit-vectors to break symmetries. Implementing integer relation operators directly on bit-vectors and providing a bit-vector implementation of alldifferent would allow us to totally eliminate integer channeling, potentially improving runtimes. Alternatively, it may be possible to define an integer view for bit-vectors which could use the integer-based alldifferent without needing a separate integer variable.

Finally, although the core of Gecode is heavily optimized, our limited C++ experience likely leaves room for the use of more advanced C++ tricks to further improve the performance of our bit-vector components.

Bibliography

[1] S. E. Anderson. *Bit Twiddling Hacks*. URL: `https://graphics.stanford.edu/~seander/bithacks.html` (visited on 2014-07-10).

[2] E. Biham and A. Shamir. "Differential Cryptanalysis of DES-like Cryptosystems." English. In: *Advances in Cryptology-CRYPT0 90*. Ed. by A. Menezes and S. A. Vanstone. Vol. 537. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1991, pp. 2–21.

[3] J. Cavanagh. *X86 Assembly Language and C Fundamentals*. Taylor & Francis, 2013.

[4] D. Coppersmith. "The Data Encryption Standard (DES) and its strength against attacks." In: *IBM Journal of Research and Development* 38.3 (May 1994), pp. 243–250.

[5] N. Dale and J. Lewis. *Computer Science Illuminated*. Jones & Bartlett Learning, 2012.

[6] *Gecode: generic constraint development environment*. 2014. URL: `http://www.gecode.org/` (visited on 2014-04-02).

[7] *Intel 64 and IA-32 Architectures Software Developer Manual*. Vol. 2A. Intel, 3:92–3:97.

[8] M. Matsui. "Linear Cryptanalysis Method for DES Cipher." English. In: *Advances in Cryptology EUROCRYPT 93*. Ed. by T. Helleseth. Vol. 765. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1994, pp. 386–397.

[9] L. D. Michel and P. Hentenryck. "Constraint Satisfaction over Bit-Vectors." In: *Principles and Practice of Constraint Programming – CP 2012*. Ed. by M. Milano. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 527–543.

[10] L. Nyhoff. *Programming in C++ for Engineering and Science*. Taylor & Francis, 2012.

[11] V. Ramamoorthy et al. "Symmetries of Nonlinearity Constraints." In: *The 11th International Workshop on Symmetry in Constraint Satisfaction Problems*. 2011.

[12] V. Ramamoorthy et al. "The Design of Cryptographic S-Boxes Using CSPs." In: *Principles and Practice of Constraint Programming – CP 2011*. Ed. by J. Lee. Vol. 6876. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 54–68.

[13] C. Schulte. “Course Notes Constraint Programming (ID2204).” In: (2012).

[14] C. Schulte. “Programming variables.” In: *Modeling and Programming with Gecode*. Ed. by C. Schulte, G. Tack, and M. Z. Lagerkvist. Corresponds to Gecode 4.2.1. 2013.

[15] C. Schulte and G. Tack. “Views and Iterators for Generic Constraint Implementations.” In: *Recent Advances in Constraints*. Ed. by B. Hnich et al. Vol. 3978. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 118–132.

[16] C. Schulte and G. Tack. “Programming propagators.” In: *Modeling and Programming with Gecode*. Ed. by C. Schulte, G. Tack, and M. Z. Lagerkvist. Corresponds to Gecode 4.2.1. 2013.

[17] C. Schulte, G. Tack, and M. Z. Lagerkvist. “Modeling.” In: *Modeling and Programming with Gecode*. Ed. by C. Schulte, G. Tack, and M. Z. Lagerkvist. Corresponds to Gecode 4.2.1. 2013.

[18] C. E. Shannon. “Communication Theory of Secrecy Systems.” In: *Bell System Technical Journal* 28.4 (1949), pp. 656–715.

[19] P. Silvester and D. Lowther. *Computer Engineering: Circuits, Programs, and Data*. Oxford University Press, 1989.

[20] D. Stinson. *Cryptography: Theory and Practice, Third Edition*. Discrete Mathematics and Its Applications. Taylor & Francis, 2005.