Finding Optimal Bitsliced Implementations of 4×4-bit S-boxes*

Markus Ullrich**, Christophe De Cannière, Sebastiaan Indesteege, Özgül Küçük, Nicky Mouha***, and Bart Preneel

[1] Department of Electrical Engineering ESAT/SCD-COSIC, Katholieke Universiteit Leuven. Kasteelpark Arenberg 10, B-3001 Heverlee, Belgium.
[2] Interdisciplinary Institute for BroadBand Technology (IBBT), Belgium.
Markus.Ullrich@esat.kuleuven.be

Abstract. In this paper, we will present an approach to find efficient bitsliced implementations of invertible 4×4-bit s-boxes. The approach generalises the methods introduced by Osvik [12]. We consider equivalence classes of s-boxes under linear and affine equivalence and search for the most efficient s-box in each class. The properties of these s-boxes are discussed and compared with s-boxes used in existing cryptographic primitives. Finally, we propose a methodology to design efficient cryptographic primitives by making use of our findings.

Keywords: Cryptography, s-boxes, efficiency, bitslicing, software implementation, equivalence class.

1 Introduction

The integration of security applications on embedded and mobile platforms requires lightweight and efficient cryptographic primitives. In this work, we focus on the efficient software implementation of 4×4-bit s-boxes. S-boxes are an essential part of many cryptographic primitives. We investigate bitsliced implementations, which proved to be a very efficient implementation technique for AES [5, 7, 6] and Serpent [12]. Bitslicing is an implementation technique where bitwise operations can be parallelised due to the fact that processors work with registers larger than one bit. On modern CPUs this can be up to a factor of 128 [11]. Intel's upcoming AVX extension will even allow 256-bit operands.

* This work was supported in part by the Research Council K.U.Leuven: GOA TENSE, and by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy), and in part by the European Commission through the ICT program under contract ICT-2007-216676 ECRYPT II.
** This author is an FWO Doctoral researcher.
*** This author is funded by a research grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

In this paper, we show how to find the most efficient bitsliced implementations of s-boxes. In contrast to previous approaches, the aim of this work is to cover the whole range of 4×4-bit s-boxes by classifying the s-boxes into equivalence classes and finding the most efficient s-box per class. The classification criterion we choose provides invariant properties with regard to linear and differential cryptanalysis. We discuss the s-box's application in the design of efficient ciphers.

In Sect. 2, a brief overview of previous work is presented. The classification method and the search for efficient implementations are presented in Sect. 3. In Sect. 4, we discuss the results obtained and compare them to s-boxes used in other cryptographic primitives. A new approach to design primitives is proposed in Sect. 5, and it is explained how designers can benefit from this work. In Sect. 6, we suggest further ideas to be investigated based on our findings. We conclude in Sect. 7.

2 Previous Work

Osvik introduces algorithms to find efficient implementations of a given s-box in [12]. The approach requires that the s-box is given and does not search for s-boxes with optimal implementations costs. Osvik's approach uses a heuristic algorithm. While this reduces the search complexity, it does not guarantee that the optimal solution will be found. He applied these techniques to find efficient implementations of the Serpent s-boxes.

Watanabe used a search technique similar to Osvik's to generate an efficient s-box for the hash function Luffa [14]. In contrast to Osvik's approach, the s-boxes were not specified in advance. The aim of the search is to find an s-box with strong properties and a low implementation cost. In a first version, the s-boxes have been generated using predefined building blocks. A first version of building blocks guaranteed invertibility and the second version has been introduced to improve the performance. The second version of the Luffa s-box has been designed by combining random instructions. This method is expected to find good s-boxes, but it is not guaranteed that an s-box with the optimal trade-off between the implementation cost and the cryptographic properties will be found.

3 Our Approach

3.1 S-box Properties and Classification

Typically, block ciphers are constructed using non-linear s-boxes and linear layers. The decomposition into linear and non-linear building blocks

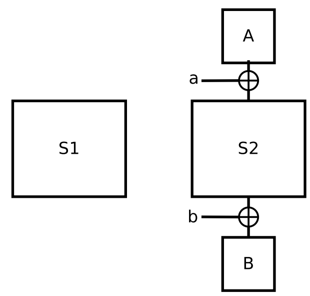


Fig. 1. Affine equivalence algorithm

is not unique. Linear operations can be moved between those building blocks. Any linear or affine transformation applied to the input or the output of an s-box could also be incorporated in the linear layer instead [9].

Definition 1. *S-boxes are said to be affinely equivalent if the following relation holds:*

$$S_1(x) = B \cdot (S_2(A \cdot x \oplus a) \oplus b), \ \ \forall x \in \{0, 1\}^n \ \ , \tag{1}$$

with invertible linear mappings (A, B) and constants (a, b). The relation is illustrated in Fig. 1.

The most important property is that all members of an affine equivalence class share the same properties with respect to linear and differential cryptanalysis. For this reason, the affine equivalence has been used as a classification criterion in our search.

In order to characterise s-boxes, we make use of a set of important properties:

- Difference distribution table [2]
- Linear approximation table [8]
- Branch number
- Existence of fixed points
- Algebraic degree of the output bits

The difference distribution table indicates the probability that a given input difference results in a certain output difference. The probability that certain correlations hold between inputs and outputs is given by the linear approximation table. Worst-case probabilities are important to bound the complexity of cryptanalytic attacks. Respectively, these probabilities are referred to as the maximal differential probability (MDP) and maximal linear probability (MLP). The linear approximation table and difference distribution table depend only on the s-box. Any affine transformation applied to the s-box can only swap elements in these tables, but not change their values.

Another property is the branch number. It is an important property describing the diffusion capabilities.

Definition 2. *In this paper, the branch number is defined as*

$$B = min_{a,b\neq a}(w_h(a \oplus b) + w_h(S(a) \oplus S(b)) \ , \tag{2}$$

where w_h is the Hamming weight and S the s-box.

A different definition is used in [4] where the branch number is defined as a measure of how many s-boxes will be active in the next round. This second definition is more suitable for entire rounds of a cipher than for single s-boxes. The branch number according to Def. 2 depends on the position of the values in the difference distribution table. This implies that it can be influenced by affine transformations.

Fixed points depend on the affine transformations, due to affine constants which can create or avoid any fixed point.

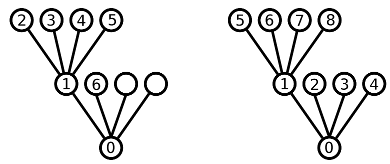
The affine transformation has only limited influence on the polynomial representation of the output bits. It can merge or cancel out monomials by performing additions between the output bits or change the operands by performing additions at the input side. However, it can never create new monomials of higher degree than the maximum degree already present. It is also not possible to cancel out all maximum degree monomials by affine transformations. The maximal algebraic degree is thus invariant within each class.

3.2 Introduction into Search Algorithms

Before presenting our search approach, a brief introduction to search terminology and algorithms has to be given.

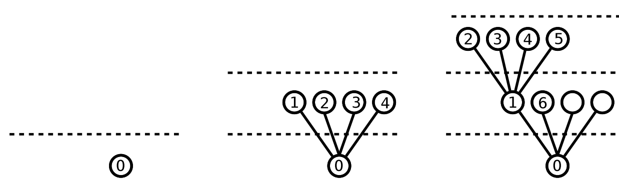
Definition 3. *The branching factor is defined as the average number of childnodes of a node.*

Depth first search (DFS) is an algorithm to search trees. It starts at the tree's root and visits as many nodes of one branch as possible before backtracking, see Fig. 2(a). The time complexity of DFS is exponential in the depth, $\mathcal{O}({b_t}^d)$ (with b_t the tree's branching factor and d the depth). The algorithm's memory requirement is linear in the depth, $\mathcal{O}(d)$. Only



(a) DFS (b) BFS

Fig. 2. The depth first search (DFS) and the breadth first search (BFS)



(a) Limit 0 (b) Limit 1 (c) Limit 2

Fig. 3. Iterative deepening depth first search (ID-DFS)

the nodes on the path to the current node have to be held in memory. In infinite trees the depth first search will not terminate.

The breadth first search (BFS) visits all nodes of a depth before advancing to higher level of depth. It finds the solution which is the closest to the root first. The main disadvantage of the search is the high memory requirement which is exponential in the depth, $\mathcal{O}({b_t}^d)$. The time complexity is $\mathcal{O}({b_t}^d)$ as for the DFS.

In the case of an unknown depth of the first solution the iterative deepening depth first search (ID-DFS) solves the previously discussed problem of the DFS. The algorithm repeatedly runs depth limited depth first searches while increasing the limit. The limit is incremented by the minimal possible increment such that at least one more node will be visited in the next run. The algorithm is illustrated in Fig. 3. The ID-DFS combines the property of monotonously increasing the depth, like in BFS, with the lower space complexity of the DFS. The fact that many nodes are visited multiple times does not influence the efficiency as much as it may seem. The higher the branching factor the more significant become the nodes that are further away from the root. The time and space complexity are the same as for the DFS.

3.3 Search

The general goal of this work is to find the most efficient implementations of s-boxes. In Sect. 3.1, we introduced a classification criterion so that we can group s-boxes with common properties with respect to linear and differential cryptanalysis. Combined, one can find the most efficient implementation that fulfil certain properties. In order to find the implementation, we introduce a search method.

The basic idea during the search is to go through all possible combinations of an instruction set. The architecture, for which the implementations are searched for, consists of 5 registers and an instruction containing the following instructions: *AND*, *OR*, *XOR*, *NOT*, *MOV* (copy). Four registers are needed to store the information of a 4×4-bit s-box and a fifth register is used for intermediate storage for the calculation of invertible s-boxes. Our choice of the instruction set consists of the basic operations of the commonly used architectures.

An important property of the search is that the number of instructions is monotonously increasing. The first node found that represents a new class is therefore also an optimal implementation of that class with respect to the number of instructions.

Another property of the search is its deterministic behaviour. Determinism guarantees that each run of the software will find exactly the same set of representatives with the following definition:

Definition 4. *The representative is the s-box of an affine equivalence class with the least implementation cost (number of instructions) and among those of equal cost, the order of the instructions determines which is chosen.*

These properties can be achieved by an iterative deepening depth first search (ID-DFS).

Using the previously defined architecture results in a branching factor of 85, which is the number of instruction and operand combinations. Reaching high depths with such a branching factor is not feasible. Therefore, we implemented a set of rules that remove redundant nodes from the search tree. First, we used the non-heuristic rules from Osvik [12]. They consist of the following rules:

- Recursion stops when the register contents can no longer generate a permutation.
- When two instruction sequences are identied as being equivalent, we remove one of them from the search.

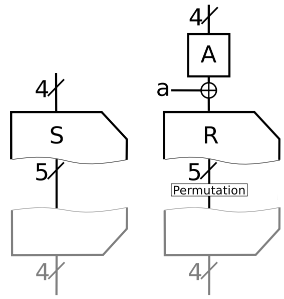


Fig. 4. Affine equivalence used for caching

- No instruction other than *MOV* may make a register contain a copy of the value in another register.
- Unread registers may not be written to by the *MOV* instruction.
- Negated registers (those last modied by a *NOT* instruction) are marked as such, and may not again be negated until they have been read.

Second, we also removed nodes that are affinely equivalent with other states saved in a cache. We interpreted the five registers of a node as an unfinished 4×5-bit s-box. We consider the s-box as incomplete because not every node represents an invertible s-box. Some nodes are only intermediate steps before reaching a valid s-box. The affine mapping at the output side can thus not be applied. More instructions may follow. Only bit swapping is allowed at this stage because the order of bits is of no impact. The equivalence algorithm is illustrated in Fig. 4. S and R are affinely equivalent if there exists invertible linear mappings (A, B) and a constant (a) such that the two s-boxes become equal. The branching factor was reduced from 85 initially down to 10 when applying Osvik's rules. By using affine equivalences in the caching algorithm, the branching factor could be even decreased further to less than 7.

3.4 Restrictions of the Search

We restricted the search by imposing the following properties:

- The size of the s-boxes has been set to 4×4-bit. 4×4-bit s-boxes are the smallest commonly used s-boxes. The number of affine classes was also important for this decision. While 4×4-bit s-boxes have 302 affine equivalence classes, 5×5-bit s-boxes have about 2^{61} classes [9]. The latter is impractical to enumerate.

– We decided to limit our scope to invertible s-boxes. In principle, the approach would work with non-invertible s-boxes as well. The main argument is the efficiency of the search. The search space for non-invertible s-boxes is larger and some of the used algorithms, e.g., the equivalence algorithm, are less efficient for non-invertible s-boxes.

4 Results

The search was designed such that it would eventually find implementations for all classes. Correct results are found earlier so that the search can be stopped at any time. The search ran for more than 2 months on a computer with 8 cores[3] and 64 Gb of RAM used for caching. At the moment the search was stopped, our program finished searching s-boxes with 12 instructions. The search for s-boxes with 13 instructions is incomplete. The search found the most efficient representatives of 272 out of 302 equivalence classes. A list of these representatives is presented in appendix A. These representatives represent about 90% of all 4×4-bit s-boxes. There are two main reasons for stopping the search. First, the most efficient s-box for a class with optimal non-linear properties, MDP $= 1/4$ and MLP $= 1/2 + 1/4$, has been found. This s-box makes use of only 9 instructions. S-boxes with 13 instructions have 44% higher cost but their advantage against linear and differential cryptanalysis is limited. Second, the complexity of the search is exponential in the depth of the search. Much more resources are required to find the last equivalence classes.

4.1 Cost vs. Properties

As a first analysis, we look at the correlation between the implementation cost and the properties regarding linear and differential cryptanalysis. We focus on the worst case probability of linear approximations or differentials, i.e., MLP and MDP. Tables 1 and 2 show the minimum required number of instructions to obtain a certain MLP or MDP. It can be seen that 9 instructions is an important threshold. With fewer instructions, the s-box has linear approximations and differentials with non-optimal probabilities. For fewer than 9 instructions there always exist correlations of 1 between certain input and output bits. Using more instructions, on the other hand, one can not improve the worst case probability, but reduce the number of differentials or linear approximations that have the worst case probability.

[3] Intel Xeon CPU X7350 @ 2.93 GHz

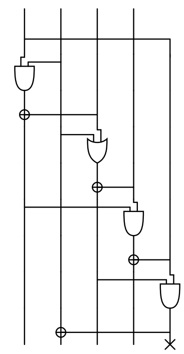


Fig. 5. Representative of class 13

Table 1. Minimum number of instructions required to implement an s-box with a given MLP. MLP−1/2 is the probability bias and $|c|$ is the correlation.

MLP $-1/2$	1/8	1/4	3/8	1/2
$\lvert c \rvert$	1/4	1/2	3/4	1
min. cost	-	9	9	0

In [9], 16 affine equivalence classes are presented that have optimal non-linear properties. The optimal properties are MDP = 1/4 and MLP = 1/2 + 1/4. Class 13 (see appendix A) has the least implementation cost among the optimal classes. The representative of class 13 is shown in Fig. 5.

Another interesting property is the branch number. All representatives that have been found have the same branch number. In fact, the branch numbers are of the smallest possible value of 2. The representatives can be considered as the essence of an s-box that achieves certain non-linear properties. However, all found representatives are weak with regard to linear mixing properties. This can be compensated for by the linear/affine layer of a cipher. It can be shown that there exist invertible linear mappings that translate any arbitrary differential at the input and the output to differentials with weight 1. Thus, every linear equivalence class and consequently also every affine equivalence class contains s-boxes with branch number 2.

Table 2. Minimum number of instructions required to implement an s-box with a given MDP

MDP	1/8	1/4	3/8	1/2	5/8	3/4	7/8	1
min. cost	-	9	10	6	9	6	-	0

Table 3. The 4×4-bit s-boxes used in some symmetric primitives

Primitive	S-box	Class
Serpent [1]	S_4, S_5	9
	S_4^{-1},S_5^{-1}	10
	S_0^{-1},S_1	14
	S_0,S_1^{-1}	15
	S_2,S_2^{-1},S_6,S_6^{-1}	16
	S_3,S_3^{-1},S_7,S_7^{-1}	not found
Luffa [14]	Q	16
Noekeon [3]	$S = S^{-1}$	13

4.2 Affine Equivalence and the NOT Instruction

We observed that none of the found representatives make use of the NOT instruction. This implies that the optimal implementations of all of them have zero as fixed point. The search finds one representative per class. It can not be excluded that an equivalence class does contain another s-box with the same number of instruction but without or with a different fixed point. For class 13, it is interesting to notice that the optimal implementation without fixed point found by Watanabe [14] needs 10 instructions instead of 9. The s-boxes were not equal, but affinely equivalent to each other. One has to notice that the optimisation processes that are compared have different target platforms.

Conjecture 1. We conjecture that there exist optimal implementations for all 302 equivalence classes that do not make use of the NOT instruction.

For s-boxes that are designed with 5 registers and a structure that resembles an unbalanced Feistel network [13], we have found many examples showing that any NOT instruction can be moved using De Morgan's laws to either of the ends of the s-box. We call an s-box 'Feistel resembling' if the outcome of a non-linear operation of maximum 4 registers is added to another register. The affine equivalence makes NOT instructions at the ends irrelevant because the affine constants can compensate them. For the s-boxes that are not of this type no simple explanation has been found. We leave a formal proof to further research.

4.3 Comparison with Known Primitives

Furthermore, we have classified the s-boxes of Serpent, Luffa and Noekeon [1, 3, 10]. Even though they are in classes for which we have found most efficient implementations, none of them is equal to a representative. The reason for this is assumed to be the design strategy and the properties that are variant within a class, like the involution in the case of Noekeon. We will explain the reason for this in more detail in Sect. 5.

Interesting in this table is that Noekeon uses an s-box from class 13. Noekeon is a cipher designed for efficient bitsliced implementation, and class 13 is the class for which we found the fastest bitsliced implementation. However, from knowing the cost of a representative one can not conclude about the cost of any other s-box within the class. Second, Noekeon requires that its s-box is an involution. This is only possible in some of the classes for which the inverse s-box is member of the same class, e.g., classes 11 and 13.

5 A New Design Approach

In the previous section, we compared the s-boxes used in existing primitives with the representatives found in this work. It turned out that none of the primitives used one of the representatives. Furthermore, it is not possible to replace the s-boxes by one of the representatives because of the design goals set by the designers. These design goals involve properties that are variant within affine equivalence classes.

The design strategies for many primitives in use separate the design of the various components from the s-box design. During the process of designing the other parts, the specifications of the s-box are refined step by step. Some of these refinements restrict properties that are variant within a class. The table indicating the best implementations can only be used if the specifications for the s-box contain only properties that are invariant. Any of the variant properties is restricting the choice of which s-box of the class has to be used and may cause that the most efficient member, the representative, can not be chosen.

To benefit efficiently from the representatives, one has to follow a new approach when designing the primitives. In the first step, the s-box is chosen. For this purpose, the designer needs target platform specific tables of most efficient representatives. One of the representatives will be chosen depending on the non-linear properties that are desired. It is important to choose the constraints with care. Some of the s-boxes have the same histogram of the linear approximation table and the differential distribution table but different implementation costs. After the s-box has been chosen the linear layer is designed in such a way that all the design goals of the cipher are fulfilled.

It is expected that this methodology leads to a cipher with low implementation cost. We concentrate on the search of s-box implementations and leave the application of our results for cipher design for future research.

6 Future Work

- **Verifying the new design approach.** In Sect. 5, we introduced a new design methodology for cryptographic primitives. The problem of the interaction between the linear and the substitution layer has not been fully investigated. We suggest to investigate if using the most efficient s-boxes can lead to highly efficient primitives and that possible shortcomings of the s-boxes regarding variant properties can be compensated by the linear layer in an efficient way.
- **Generalising the findings to design larger s-boxes.** This work focused on the design of 4×4-bit s-boxes. The application of the methods for larger s-boxes is assumed to be not feasible. Therefore, we suggest to analyse if it is possible to generalise the findings of 4×4-bit s-boxes to larger s-boxes.
- **Proof for NOT instruction.** We suggest to further investigate the fact that not a single one of the representatives found, makes use of the NOT instruction. We conjecture that this is the case for all representatives of the affine equivalence classes. A proof is left to future research.
- **Extended search.** The search targets a basic architecture. Modern processors offer more advanced techniques, such as parallelism, pipelining and instruction set extensions. Using these features for s-box implementations can result in other tradeoffs which could be investigated.

7 Conclusion

In this paper, we presented an approach to find efficient bitsliced implementations of invertible 4×4-bit s-boxes. In a first step, we introduced a search to find the most efficient implementations of s-boxes.

We presented a list (see appendix A) of optimal implementations for a specified instruction set. This list contains the most efficient classes covering 90% of all 4×4-bit s-boxes.

The representatives have been analysed for their properties and were compared with s-boxes used in known primitives. Finally, we presented a new design methodology for cryptographic primitives.

References

1. R. J. Anderson, E. Biham, and L. R. Knudsen. The case for Serpent. In *AES Candidate Conference*, pages 349–354, 2000.
2. E. Biham and A. Shamir. Differential cryptanalysis of DES-like cryptosystems. In *CRYPTO '90: Proceedings of the 10th Annual International Cryptology Conference on Advances in Cryptology*, volume 537 of *LNCS*, pages 2–21. Springer, 1991.
3. J. Daemen, M. Peeters, G. Van Assche, and V. Rijmen. NESSIE proposal: NOEKEON. Submitted as an NESSIE Candidate Algorithm, `http://www.cryptonessie.org`, 2000.
4. J. Daemen and V. Rijmen. *The Design of Rijndael*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.
5. E. Käsper and P. Schwabe. Faster and timing-attack resistant AES-GCM. In C. Clavier and K. Gaj, editors, *CHES*, volume 5747 of *LNCS*, pages 1–17. Springer, 2009.
6. R. Könighofer. A fast and cache-timing resistant implementation of the AES. In *CT-RSA*, volume 4964 of *LNCS*, pages 187–202. Springer, 2008.
7. M. Matsui and J. Nakajima. On the power of bitslice implementation on Intel Core2 processor. In *CHES*, volume 4727 of *LNCS*, pages 121–134. Springer, 2007.
8. M. Matsui and A. Yamagishi. A new method for known plaintext attack of FEAL cipher. In *EUROCRYPT*, volume 658 of *LNCS*, pages 81–91, 1992.
9. C. De Cannière. *Analysis and design of symmetric encryption algorithms*. PhD thesis, K.U. Leuven, 2007.
10. C. De Cannière, H. Sato, and D. Watanabe. Hash function Luffa: Specification. Submission to NIST (Round 2), 2009.
11. Intel Corporation. Intel(R) C++ compiler for Linux* intrinsics reference. URL: `http://software.intel.com/file/6373`, 2006.
12. D. A. Osvik. Speeding up Serpent. In *AES Candidate Conference*, pages 317–329, 2000.
13. B. Schneier and J. Kelsey. Unbalanced feistel networks and block cipher design. In D. Gollmann, editor, *FSE*, volume 1039 of *LNCS*, pages 121–144. Springer, 1996.
14. D. Watanabe. How to generate the Sbox of Luffa. Early Symmetric Crypto Seminar, ESC2010, January 2010.

A List of Representatives

Tables 4–10 list the representatives of the affine equivalence classes. The following properties are included: the linear histogram (c), the differential histogram (p), the implementation cost, the branch number (bn) of the representative, the maximum algebraic degree of the output bits (deg) and the equivalence class that contains the inverse of the s-box (inv).

Table 4. Implementations of the affine equivalence classes

	Representative	$\lvert c\rvert = 1/4$	1/2	3/4	1	$p = 1/8$	1/4	3/8	1/2	5/8	3/4	7/8	1	cost	bn	deg	inv
1	?	120	30	0	1	90	15	0	0	0	0	0	1	?	?	3	?
2	?	120	30	0	1	90	15	0	0	0	0	0	1	?	?	3	?
3	?	120	30	0	1	90	15	0	0	0	0	0	1	?	?	3	?
4	?	120	30	0	1	90	15	0	0	0	0	0	1	?	?	3	?
5	?	120	30	0	1	90	15	0	0	0	0	0	1	?	?	3	?
6	?	120	30	0	1	90	15	0	0	0	0	0	1	?	?	3	?
7	?	120	30	0	1	90	15	0	0	0	0	0	1	?	?	3	?
8	?	120	30	0	1	90	15	0	0	0	0	0	1	?	?	3	?
9	`0cabf9d4e8635172`	112	32	0	1	84	18	0	0	0	0	0	1	11	2	3	10
10	`01298bd7cfe654a3`	112	32	0	1	84	18	0	0	0	0	0	1	12	2	3	9
11	`0a43562edfb1c789`	112	32	0	1	84	18	0	0	0	0	0	1	13	2	3	11
12	?	112	32	0	1	84	18	0	0	0	0	0	1	?	?	3	?
13	`086d5f7c4e2391ba`	96	36	0	1	72	24	0	0	0	0	0	1	9	2	3	13
14	`086c7e5f4d21b39a`	96	36	0	1	72	24	0	0	0	0	0	1	10	2	3	15
15	`0845d7fec6a391b2`	96	36	0	1	72	24	0	0	0	0	0	1	10	2	3	14
16	`01a2987cdef4563b`	96	36	0	1	72	24	0	0	0	0	0	1	11	2	3	16
17	?	120	30	0	1	93	12	1	0	0	0	0	1	?	?	3	?
18	`02839b7eca65df14`	112	32	0	1	87	15	1	0	0	0	0	1	12	2	3	20
19	`04afb6372e81c95d`	112	32	0	1	87	15	1	0	0	0	0	1	12	2	3	?
20	`02415f3e8bc6a9d7`	112	32	0	1	87	15	1	0	0	0	0	1	12	2	2	18
21	`0251c6afd7984e3b`	112	32	0	1	87	15	1	0	0	0	0	1	13	2	3	21
22	?	112	32	0	1	87	15	1	0	0	0	0	1	?	?	3	?
23	?	120	30	0	1	96	9	2	0	0	0	0	1	?	?	3	?
24	?	120	30	0	1	96	9	2	0	0	0	0	1	?	?	3	?
25	`0c69735248af1dbe`	96	36	0	1	78	18	2	0	0	0	0	1	11	2	2	26
26	`06a953b842c7df1e`	96	36	0	1	78	18	2	0	0	0	0	1	11	2	2	25
27	`0a2387bfc5de4961`	96	36	0	1	78	18	2	0	0	0	0	1	12	2	2	29
28	`0a2387bf4d56c1e9`	96	36	0	1	78	18	2	0	0	0	0	1	12	2	2	28
29	`0913a4bf2e6587dc`	96	36	0	1	78	18	2	0	0	0	0	1	12	2	3	27
30	`06af7d5e48c391b2`	96	36	0	1	81	15	3	0	0	0	0	1	11	2	3	30
31	`04598ceb6a72f3d1`	96	36	0	1	80	18	0	1	0	0	0	1	11	2	3	31
32	`08a319f4c6e5d7b2`	64	44	0	1	64	24	0	2	0	0	0	1	9	2	3	33
33	`086d5f7e4c2193ba`	64	44	0	1	64	24	0	2	0	0	0	1	9	2	2	32
34	?	119	28	1	1	78	21	0	0	0	0	0	1	?	?	3	?
35	?	119	28	1	1	78	21	0	0	0	0	0	1	?	?	3	?
36	?	119	28	1	1	78	21	0	0	0	0	0	1	?	?	3	?
37	`03298bd5efc476a1`	111	30	1	1	72	24	0	0	0	0	0	1	11	2	3	37
38	`03d74f985ec621ab`	111	30	1	1	72	24	0	0	0	0	0	1	12	2	3	38
39	`0e8952d7ca4b61f3`	119	28	1	1	81	18	1	0	0	0	0	1	13	2	3	39
40	`0283db4eca769f15`	119	28	1	1	81	18	1	0	0	0	0	1	13	2	3	40
41	?	119	28	1	1	81	18	1	0	0	0	0	1	?	?	3	?
42	?	119	28	1	1	81	18	1	0	0	0	0	1	?	?	3	?
43	`0c2784fa5961e3db`	111	30	1	1	75	21	1	0	0	0	0	1	12	2	2	44
44	`0c4d9fba8e635172`	111	30	1	1	75	21	1	0	0	0	0	1	12	2	3	43
45	`06abc84e79f2d153`	111	30	1	1	75	21	1	0	0	0	0	1	12	2	3	45

Table 5. Implementations of the affine equivalence classes *(continued)*

	Representative	$\lvert c\rvert = 1/4$	1/2	3/4	1	$p = 1/8$	1/4	3/8	1/2	5/8	3/4	7/8	1	cost	bn	deg	inv
46	`0d9163e5fb7ac842`	119	28	1	1	84	15	2	0	0	0	0	1	13	2	3	?
47	?	119	28	1	1	84	15	2	0	0	0	0	1	?	?	3	?
48	?	119	28	1	1	84	15	2	0	0	0	0	1	?	?	3	?
49	?	119	28	1	1	84	15	2	0	0	0	0	1	?	?	3	?
50	?	119	28	1	1	84	15	2	0	0	0	0	1	?	?	3	?
51	`0283db7eca659f14`	111	30	1	1	78	18	2	0	0	0	0	1	11	2	3	63
52	`0285cf4b9a36de71`	111	30	1	1	78	18	2	0	0	0	0	1	11	2	3	59
53	`0c3e97af86d4512b`	111	30	1	1	78	18	2	0	0	0	0	1	12	2	3	70
54	`038a75dbcf6e1294`	111	30	1	1	78	18	2	0	0	0	0	1	12	2	3	54
55	`038a64dbcf7e1295`	111	30	1	1	78	18	2	0	0	0	0	1	12	2	3	55
56	`0481e37d6afbc952`	111	30	1	1	78	18	2	0	0	0	0	1	12	2	3	57
57	`04987bcf6ad251e3`	111	30	1	1	78	18	2	0	0	0	0	1	12	2	3	56
58	`0c2db39a6e857f14`	111	30	1	1	78	18	2	0	0	0	0	1	12	2	3	58
59	`086e7d5c4f21b39a`	111	30	1	1	78	18	2	0	0	0	0	1	12	2	3	52
60	`0cf1634b9d25a78e`	111	30	1	1	78	18	2	0	0	0	0	1	12	2	3	66
61	`0a24193685def7bc`	111	30	1	1	78	18	2	0	0	0	0	1	12	2	3	62
62	`0a23486519dcfb7e`	111	30	1	1	78	18	2	0	0	0	0	1	12	2	3	61
63	`04f28d617be3c95a`	111	30	1	1	78	18	2	0	0	0	0	1	13	2	3	51
64	`0cfbae138594726d`	111	30	1	1	78	18	2	0	0	0	0	1	13	2	3	?
65	`0debaf129584736c`	111	30	1	1	78	18	2	0	0	0	0	1	13	2	3	65
66	`07d9af54bec86231`	111	30	1	1	78	18	2	0	0	0	0	1	13	2	3	60
67	`0ae36592748cdf1b`	111	30	1	1	78	18	2	0	0	0	0	1	13	2	3	?
68	`086293efc7b5d4a1`	111	30	1	1	78	18	2	0	0	0	0	1	13	2	3	73
69	`04816aced372f95b`	111	30	1	1	78	18	2	0	0	0	0	1	13	2	3	69
70	`0281df5bce679a34`	111	30	1	1	78	18	2	0	0	0	0	1	13	2	3	53
71	`0182cf6ade579b34`	111	30	1	1	78	18	2	0	0	0	0	1	13	2	3	71
72	`0283db7fca659e14`	111	30	1	1	78	18	2	0	0	0	0	1	13	2	3	?
73	`0243d6aec7b95f18`	111	30	1	1	78	18	2	0	0	0	0	1	13	2	3	68
74	?	111	30	1	1	78	18	2	0	0	0	0	1	?	?	3	?
75	?	111	30	1	1	78	18	2	0	0	0	0	1	?	?	3	?
76	?	111	30	1	1	78	18	2	0	0	0	0	1	?	?	3	?
77	`0bf36482759cde1a`	111	30	1	1	81	15	3	0	0	0	0	1	13	2	3	77
78	?	111	30	1	1	81	15	3	0	0	0	0	1	?	?	3	?
79	`08e42ac1d7f5b396`	95	34	1	1	72	18	4	0	0	0	0	1	11	2	3	80
80	`04693fd17b52eac8`	95	34	1	1	72	18	4	0	0	0	0	1	11	2	3	79
81	`09e65cf74d82ba31`	95	34	1	1	72	18	4	0	0	0	0	1	11	2	2	82
82	`0c6bd9f2e8a51734`	95	34	1	1	72	18	4	0	0	0	0	1	11	2	3	81
83	`08c52ae1d4f6b397`	95	34	1	1	72	18	4	0	0	0	0	1	12	2	3	83
84	`04ae8c219fbd5376`	63	42	1	1	78	0	14	0	0	0	0	1	10	2	3	84
85	`0dbea6372f91c845`	111	30	1	1	80	18	0	1	0	0	0	1	12	2	2	85
86	`0ea62c93f4d587b1`	111	30	1	1	80	18	0	1	0	0	0	1	13	2	3	86
87	`0913b2c486ed5a7f`	118	26	2	1	72	21	2	0	0	0	0	1	13	2	3	87
88	`0d7c2a186e5fb349`	118	26	2	1	72	21	2	0	0	0	0	1	13	2	3	89
89	`0c6749be1532f8da`	118	26	2	1	72	21	2	0	0	0	0	1	13	2	3	88
90	`0e2f84acb7d65319`	118	26	2	1	72	21	2	0	0	0	0	1	13	2	3	90

Table 6. Implementations of the affine equivalence classes *(continued)*

	Representative	$\lvert c\rvert = 1/4$	1/2	3/4	1	$p = 1/8$	1/4	3/8	1/2	5/8	3/4	7/8	1	cost	bn	deg	inv
91	0329d7e8f4c51ba6	118	26	2	1	72	21	2	0	0	0	0	1	13	2	3	91
92	?	118	26	2	1	72	21	2	0	0	0	0	1	?	?	3	?
93	0c2dbf16ae497358	110	28	2	1	66	24	2	0	0	0	0	1	11	2	2	95
94	095f18e4a7b3d2c6	110	28	2	1	66	24	2	0	0	0	0	1	11	2	2	94
95	04e8ca639bfd5172	110	28	2	1	66	24	2	0	0	0	0	1	12	2	3	93
96	0bd57f243a18e6c9	118	26	2	1	75	18	3	0	0	0	0	1	13	2	3	96
97	0913b24ca6ed587f	118	26	2	1	75	18	3	0	0	0	0	1	13	2	3	97
98	0425be968fdc731a	118	26	2	1	75	18	3	0	0	0	0	1	13	2	3	98
99	0724ae85bfdc6319	118	26	2	1	75	18	3	0	0	0	0	1	13	2	3	99
100	01b2c5e3f7d689a4	118	26	2	1	75	18	3	0	0	0	0	1	13	2	3	102
101	0ac5d736f4b912e8	118	26	2	1	75	18	3	0	0	0	0	1	13	2	3	101
102	09657cade4831bf2	118	26	2	1	75	18	3	0	0	0	0	1	13	2	3	100
103	0821e7ca6fd593b4	118	26	2	1	75	18	3	0	0	0	0	1	13	2	3	104
104	012bd4e3c7f598a6	118	26	2	1	75	18	3	0	0	0	0	1	13	2	3	103
105	?	118	26	2	1	75	18	3	0	0	0	0	1	?	?	3	?
106	0a387f496edcb125	110	28	2	1	69	21	3	0	0	0	0	1	11	2	3	109
107	0425bd968ecf731a	110	28	2	1	69	21	3	0	0	0	0	1	11	2	3	115
108	03298bd5cfe476a1	110	28	2	1	69	21	3	0	0	0	0	1	11	2	3	111
109	086e5c7d4f2391ba	110	28	2	1	69	21	3	0	0	0	0	1	11	2	3	106
110	06853d942cab71fe	110	28	2	1	69	21	3	0	0	0	0	1	11	2	3	112
111	0bd74f985ec621a3	110	28	2	1	69	21	3	0	0	0	0	1	12	2	3	108
112	06e915dbf37a42c8	110	28	2	1	69	21	3	0	0	0	0	1	12	2	3	110
113	0ec9731dfb52a486	110	28	2	1	69	21	3	0	0	0	0	1	12	2	3	114
114	08a1f356e24db79c	110	28	2	1	69	21	3	0	0	0	0	1	12	2	3	113
115	0942563718fdabec	110	28	2	1	69	21	3	0	0	0	0	1	12	2	3	107
116	0c8962e5fb7a41d3	118	26	2	1	78	15	4	0	0	0	0	1	13	2	3	116
117	?	118	26	2	1	78	15	4	0	0	0	0	1	?	?	3	?
118	?	118	26	2	1	78	15	4	0	0	0	0	1	?	?	3	?
119	048e26c31f957bda	110	28	2	1	72	18	4	0	0	0	0	1	12	2	3	120
120	04cae86bf1d35972	110	28	2	1	72	18	4	0	0	0	0	1	12	2	3	119
121	03a1df79ec658b24	110	28	2	1	72	18	4	0	0	0	0	1	12	2	3	124
122	0281ce6bdf549a37	110	28	2	1	72	18	4	0	0	0	0	1	12	2	3	129
123	0281ce7bdf459a36	110	28	2	1	72	18	4	0	0	0	0	1	12	2	3	128
124	0d14376cfae2958b	110	28	2	1	72	18	4	0	0	0	0	1	12	2	3	121
125	0b12756acfe4938d	110	28	2	1	72	18	4	0	0	0	0	1	12	2	3	127
126	08f5b1e42ac3d697	110	28	2	1	72	18	4	0	0	0	0	1	12	2	3	126
127	02418ae693fcd7b5	110	28	2	1	72	18	4	0	0	0	0	1	12	2	3	125
128	0bd74f91c65ea823	110	28	2	1	72	18	4	0	0	0	0	1	13	2	3	123
129	08e52ac1f4d693b7	110	28	2	1	72	18	4	0	0	0	0	1	13	2	3	122
130	08a2d5e3f6c791b4	118	26	2	1	74	21	0	1	0	0	0	1	13	2	3	130
131	0d91ea6572f3c84b	118	26	2	1	77	18	1	1	0	0	0	1	12	2	3	131
132	0481e37dfa6b59c2	110	28	2	1	71	21	1	1	0	0	0	1	12	2	3	133
133	0c86d352f74e19ba	110	28	2	1	71	21	1	1	0	0	0	1	12	2	3	132
134	0829b71ea64df35c	110	28	2	1	74	18	2	1	0	0	0	1	11	2	2	135
135	0c635172e8abf9d4	110	28	2	1	74	18	2	1	0	0	0	1	11	2	3	134

Table 7. Implementations of the affine equivalence classes *(continued)*

	Representative	$\lvert c\rvert = 1/4$	1/2	3/4	1	$p = 1/8$	1/4	3/8	1/2	5/8	3/4	7/8	1	cost	bn	deg	inv
136	`04e935fb71d86ac2`	110	28	2	1	74	18	2	1	0	0	0	1	12	2	3	137
137	`0ac9f75d1b32e684`	110	28	2	1	74	18	2	1	0	0	0	1	12	2	3	136
138	`0591e26d7afbc843`	110	28	2	1	74	18	2	1	0	0	0	1	12	2	3	138
139	`08f43bd6912c7e5a`	110	28	2	1	74	18	2	1	0	0	0	1	12	2	2	139
140	`0821f396ea45b7dc`	110	28	2	1	74	18	2	1	0	0	0	1	12	2	3	141
141	`0f415a6b97d2e8c3`	110	28	2	1	74	18	2	1	0	0	0	1	12	2	3	140
142	`0283df7ace659b14`	94	32	2	1	62	24	2	1	0	0	0	1	9	2	3	143
143	`0a6d5f7c4e2391b8`	94	32	2	1	62	24	2	1	0	0	0	1	9	2	2	142
144	`04ac8e239fbd5176`	94	32	2	1	62	24	2	1	0	0	0	1	10	2	3	145
145	`0821f6dac5e4b397`	94	32	2	1	62	24	2	1	0	0	0	1	10	2	3	144
146	`0814d5ea7f62b3c9`	94	32	2	1	62	24	2	1	0	0	0	1	10	2	3	147
147	`0425b796aec1f3d8`	94	32	2	1	62	24	2	1	0	0	0	1	10	2	2	146
148	`04617b5a8ce3d9f2`	94	32	2	1	62	24	2	1	0	0	0	1	10	2	2	150
149	`0c63d1fae82597b4`	94	32	2	1	62	24	2	1	0	0	0	1	10	2	2	149
150	`0e6bd9f2c8a51734`	94	32	2	1	62	24	2	1	0	0	0	1	10	2	3	148
151	`04af8d21be9c7356`	94	32	2	1	62	24	2	1	0	0	0	1	11	2	3	153
152	`0c81bf53d9762ea4`	94	32	2	1	62	24	2	1	0	0	0	1	11	2	3	152
153	`0a2395b8d7f6c4e1`	94	32	2	1	62	24	2	1	0	0	0	1	11	2	3	151
154	`0824f6ae5d7391cb`	94	32	2	1	62	24	2	1	0	0	0	1	11	2	3	154
155	`04639fd2e8cb517a`	94	32	2	1	64	24	0	2	0	0	0	1	10	2	3	155
156	`0c69a24ef758b31d`	117	24	3	1	66	21	4	0	0	0	0	1	12	2	3	156
157	`08296c5a4fdeb317`	117	24	3	1	66	21	4	0	0	0	0	1	13	2	3	157
158	`0bd57f26183ac4e9`	117	24	3	1	69	18	5	0	0	0	0	1	13	2	3	158
159	`08e5c7a4b391f6d2`	117	24	3	1	69	18	5	0	0	0	0	1	13	2	3	160
160	`08a9f65db217e3c4`	117	24	3	1	69	18	5	0	0	0	0	1	13	2	3	159
161	`0a8b46d2ce7f1395`	117	24	3	1	69	18	5	0	0	0	0	1	13	2	3	161
162	`08235cfae64719bd`	117	24	3	1	69	18	5	0	0	0	0	1	13	2	3	162
163	`0ac46e251b39f7d8`	109	26	3	1	63	21	5	0	0	0	0	1	11	2	3	166
164	`08e4c6a591b3d7f2`	109	26	3	1	63	21	5	0	0	0	0	1	11	2	3	165
165	`0ce53b91d7f284a6`	109	26	3	1	63	21	5	0	0	0	0	1	11	2	3	164
166	`046abec9f8d27351`	109	26	3	1	63	21	5	0	0	0	0	1	11	2	3	163
167	`0823b79ad5f64ce1`	109	26	3	1	63	21	5	0	0	0	0	1	11	2	3	169
168	`0a4e86c13bd5f792`	109	26	3	1	63	21	5	0	0	0	0	1	12	2	3	168
169	`06e842c397f5b1da`	109	26	3	1	63	21	5	0	0	0	0	1	12	2	3	167
170	`05bf8d349cae6217`	109	26	3	1	63	21	5	0	0	0	0	1	12	2	3	170
171	`0291e8a3f6d5b7c4`	117	24	3	1	72	15	6	0	0	0	0	1	12	2	3	171
172	`068cea2db7951f34`	109	26	3	1	66	18	6	0	0	0	0	1	12	2	3	172
173	`08235cfae74619bd`	109	26	3	1	66	18	6	0	0	0	0	1	12	2	3	173
174	`0219d7e3c6f48ab5`	109	26	3	1	66	18	6	0	0	0	0	1	12	2	3	174
175	`03a1ec69df748b25`	109	26	3	1	66	18	6	0	0	0	0	1	12	2	3	177
176	`0283ca6edb749f15`	109	26	3	1	66	18	6	0	0	0	0	1	12	2	3	176
177	`032476a5cfe98bd1`	109	26	3	1	66	18	6	0	0	0	0	1	12	2	3	175
178	`0289f75a6c4d3b1e`	109	26	3	1	69	15	7	0	0	0	0	1	12	2	3	178
179	`0ea3c84671f2d95b`	117	24	3	1	65	24	1	1	0	0	0	1	12	2	3	179
180	`072634bc58f9e1ad`	117	24	3	1	65	24	1	1	0	0	0	1	13	2	3	181

Table 8. Implementations of the affine equivalence classes *(continued)*

	Representative	$\lvert c\rvert = 1/4$	1/2	3/4	1	$p = 1/8$	1/4	3/8	1/2	5/8	3/4	7/8	1	cost	bn	deg	inv
181	`06b3e9c1fa4d2785`	117	24	3	1	65	24	1	1	0	0	0	1	13	2	3	180
182	`09324debf75618ac`	117	24	3	1	68	21	2	1	0	0	0	1	13	2	3	182
183	`0e42a6c3fbd97158`	109	26	3	1	62	24	2	1	0	0	0	1	11	2	3	184
184	`047bafe9d8c36251`	109	26	3	1	62	24	2	1	0	0	0	1	11	2	3	183
185	`041dbea5267fc983`	109	26	3	1	62	24	2	1	0	0	0	1	11	2	3	185
186	`0481eb7d62f3c95a`	109	26	3	1	62	24	2	1	0	0	0	1	11	2	3	186
187	`0ce53b91f7d4a286`	109	26	3	1	62	24	2	1	0	0	0	1	12	2	3	187
188	`0a3b29c5fe4d6781`	109	26	3	1	62	24	2	1	0	0	0	1	12	2	2	188
189	`0ea342c6fb78d951`	117	24	3	1	80	9	6	1	0	0	0	1	12	2	3	189
190	`0285ca4e9f36db71`	109	26	3	1	74	12	6	1	0	0	0	1	11	2	3	190
191	`06c18a4edf329b75`	109	26	3	1	74	12	6	1	0	0	0	1	11	2	3	191
192	`08e64c29d7f51b3a`	93	30	3	1	65	15	7	1	0	0	0	1	10	2	3	193
193	`08e6c4a1d7f593b2`	93	30	3	1	65	15	7	1	0	0	0	1	10	2	3	192
194	`08a1e2c6f7d4b395`	116	22	4	1	60	21	6	0	0	0	0	1	12	2	3	194
195	`0e6c84a17fd5b392`	116	22	4	1	60	21	6	0	0	0	0	1	13	2	3	195
196	`0d786e5c2a1fb349`	116	22	4	1	60	21	6	0	0	0	0	1	13	2	3	196
197	`0938e64bf75ca21d`	116	22	4	1	63	18	7	0	0	0	0	1	12	2	3	197
198	`08a1e6c3f7d4b295`	116	22	4	1	66	15	8	0	0	0	0	1	12	2	3	198
199	`08a1e6d3f7c5b294`	116	22	4	1	66	15	8	0	0	0	0	1	12	2	3	199
200	`0c6348aef71259bd`	116	22	4	1	59	24	3	1	0	0	0	1	12	2	3	200
201	`08e64c2bd7f5193a`	108	24	4	1	56	24	4	1	0	0	0	1	11	2	3	201
202	`08ce4623f5d791ba`	108	24	4	1	56	24	4	1	0	0	0	1	11	2	3	203
203	`024ce6a97f5d1b38`	108	24	4	1	56	24	4	1	0	0	0	1	11	2	3	202
204	`04a8ec23dbf95176`	108	24	4	1	56	24	4	1	0	0	0	1	11	2	3	204
205	`0ac46e2d1b397f58`	108	24	4	1	62	18	6	1	0	0	0	1	11	2	3	206
206	`08a17b95f3d2c6e4`	108	24	4	1	62	18	6	1	0	0	0	1	11	2	3	205
207	`0759aec8fbd26341`	108	24	4	1	62	18	6	1	0	0	0	1	11	2	3	208
208	`08297d5a4cefb316`	108	24	4	1	62	18	6	1	0	0	0	1	11	2	3	207
209	`02a846c397b1f5de`	108	24	4	1	62	18	6	1	0	0	0	1	12	2	3	209
210	`06ac24e397b5d1f8`	108	24	4	1	62	18	6	1	0	0	0	1	12	2	3	211
211	`0ac62e83b795f1d4`	108	24	4	1	62	18	6	1	0	0	0	1	12	2	3	210
212	`0d98ea65fb7341c2`	116	22	4	1	73	12	5	2	0	0	0	1	12	2	3	212
213	`0283de7bcf659a14`	108	24	4	1	67	15	5	2	0	0	0	1	11	2	3	214
214	`0392ce7bdf648a15`	108	24	4	1	67	15	5	2	0	0	0	1	11	2	3	213
215	`0281df7ace459b36`	92	28	4	1	48	30	0	3	0	0	0	1	9	2	3	216
216	`086f5d7e4c29b31a`	92	28	4	1	48	30	0	3	0	0	0	1	9	2	2	215
217	`0283de7bcf459a16`	115	20	5	1	56	21	6	1	0	0	0	1	12	2	3	217
218	`0829f75ae64db31c`	107	22	5	1	52	24	4	2	0	0	0	1	9	2	3	218
219	`0823b79ad5f6c4e1`	107	22	5	1	52	24	4	2	0	0	0	1	10	2	3	219
220	`08a2d5f3e7c691b4`	115	20	5	1	64	15	6	2	0	0	0	1	12	2	3	220
221	`08a2c4e597b1d3f6`	107	22	5	1	58	18	6	2	0	0	0	1	11	2	3	221
222	`08e6c4a197f5d3b2`	107	22	5	1	58	18	6	2	0	0	0	1	11	2	3	222
223	`08a64ce75df1b293`	114	18	6	1	63	6	15	0	0	0	0	1	12	2	3	223
224	`0462e8c3715bf9da`	114	18	6	1	63	6	15	0	0	0	0	1	13	2	3	224
225	`08465dbce7a291f3`	114	18	6	1	54	21	4	3	0	0	0	1	12	2	3	225

Table 9. Implementations of the affine equivalence classes *(continued)*

	Representative	$\lvert c\rvert = 1/4$	1/2	3/4	1	$p = 1/8$	1/4	3/8	1/2	5/8	3/4	7/8	1	cost	bn	deg	inv
226	08a1d5f3c4e6b297	114	18	6	1	54	21	4	3	0	0	0	1	12	2	3	226
227	0abd4ef9823657c1	106	20	6	1	54	18	6	3	0	0	0	1	10	2	2	227
228	048c62e315f97bda	114	18	6	1	69	6	9	3	0	0	0	1	13	2	3	228
229	0823d7fa4ce591b6	106	20	6	1	63	9	9	3	0	0	0	1	11	2	3	229
230	0c69a24ef718b35d	113	16	7	1	62	9	8	4	0	0	0	1	11	2	3	230
231	0938e65bf74ca21d	110	10	10	1	65	0	5	10	0	0	0	1	11	2	3	231
232	092e7456cdfb83a1	94	32	2	1	66	23	1	0	1	0	0	1	11	2	3	232
233	03fc56ed47928a1b	94	32	2	1	66	23	1	0	1	0	0	1	11	2	2	233
234	097e4d6f5c38a21b	109	26	3	1	66	23	1	0	1	0	0	1	12	2	2	234
235	06ac8e239fbd5174	92	28	4	1	60	17	7	0	1	0	0	1	10	2	3	235
236	06ac8e219fbd7354	92	28	4	1	60	17	7	0	1	0	0	1	10	2	3	236
237	08e64c29f7d51b3a	107	22	5	1	48	29	3	0	1	0	0	1	11	2	3	237
238	0921b3d5f7a86e4c	107	22	5	1	48	29	3	0	1	0	0	1	12	2	3	238
239	08e64c29d5f71b3a	107	22	5	1	54	23	5	0	1	0	0	1	11	2	3	239
240	0ac46e29f7d53b18	107	22	5	1	60	17	7	0	1	0	0	1	11	2	3	240
241	08a719b35df6e2c4	105	18	7	1	58	11	9	2	1	0	0	1	11	2	3	241
242	0a23f7d86ec591b4	105	18	7	1	58	11	9	2	1	0	0	1	11	2	3	242
243	086d5f7ec4291b3a	62	40	2	1	48	33	0	0	0	1	0	1	9	2	2	243
244	08e64c2bf7d5193a	90	24	6	1	42	27	6	0	0	1	0	1	10	2	3	244
245	084a6e1d5c397f2b	112	28	0	2	57	21	7	0	0	0	0	1	12	2	3	245
246	08ab193246cf7d5e	96	32	0	2	51	21	9	0	0	0	0	1	10	2	2	246
247	0ba981234fe6dc57	112	28	0	2	50	30	2	1	0	0	0	1	11	2	2	247
248	0c2f1d7a48693b5e	96	32	0	2	44	30	4	1	0	0	0	1	9	2	2	251
249	012b89f7cde654a3	96	32	0	2	44	30	4	1	0	0	0	1	9	2	3	249
250	082b195d4f6e7c3a	96	32	0	2	44	30	4	1	0	0	0	1	10	2	3	250
251	024513768ecbf9da	96	32	0	2	44	30	4	1	0	0	0	1	10	2	2	248
252	0c7b3e6a2f4d5918	96	32	0	2	44	30	4	1	0	0	0	1	10	2	3	252
253	086f5d7e4c293b1a	64	40	0	2	32	36	0	4	0	0	0	1	7	2	2	256
254	086f5d7e4c2391ba	64	40	0	2	32	36	0	4	0	0	0	1	7	2	2	255
255	082b197c4e6d5f3a	64	40	0	2	32	36	0	4	0	0	0	1	8	2	3	254
256	046173528cebd9fa	64	40	0	2	32	36	0	4	0	0	0	1	8	2	2	253
257	086f5d7ec4a1b392	64	40	0	2	32	36	0	4	0	0	0	1	8	2	2	257
258	08a319f6c4e7d5b2	0	56	0	2	64	0	0	14	0	0	0	1	7	2	2	258
259	09f75d26183ac4eb	110	24	2	2	45	21	11	0	0	0	0	1	11	2	3	259
260	08a357dfb192c4e6	110	24	2	2	50	18	10	1	0	0	0	1	11	2	3	260
261	08a71df395b2c4e6	94	28	2	2	40	24	8	2	0	0	0	1	9	2	3	263
262	0459afebd8c16273	94	28	2	2	40	24	8	2	0	0	0	1	9	2	3	262
263	0812b3a95d46f7ce	94	28	2	2	40	24	8	2	0	0	0	1	9	2	2	261
264	04369ca78def512b	110	24	2	2	48	24	4	3	0	0	0	1	11	2	3	264
265	032547618bacfe9d	108	20	4	2	44	12	16	1	0	0	0	1	10	2	2	265
266	08297f5a6e4d3b1c	92	24	4	2	28	30	4	5	0	0	0	1	7	2	3	267
267	082b193a4ce5f7d6	92	24	4	2	28	30	4	5	0	0	0	1	8	2	2	266
268	082b3f1a5d7e4c69	92	24	4	2	28	30	4	5	0	0	0	1	8	2	3	268
269	0461dbf28ce9537a	56	28	8	1	0	56	0	0	0	0	0	2	9	2	3	269
270	092e1436cdfb85a7	96	32	0	2	48	29	3	0	1	0	0	1	11	2	3	270

Table 10. Implementations of the affine equivalence classes *(continued)*

	Representative	$\|c\| = 1/4$	1/2	3/4	1	$p = 1/8$	1/4	3/8	1/2	5/8	3/4	7/8	1	cost	bn	deg	inv
271	`082b1a6d5f7e4c39`	96	32	0	2	48	29	3	0	1	0	0	1	11	2	3	271
272	`046f953b1d7ec82a`	110	24	2	2	36	35	3	0	1	0	0	1	11	2	3	272
273	`0a28c6e53b91f7d4`	92	24	4	2	40	17	11	2	1	0	0	1	9	2	3	273
274	`082a4ce51b39d7f6`	92	24	4	2	40	17	11	2	1	0	0	1	9	2	3	274
275	`0a28c4e53b91f7d6`	106	16	6	2	32	23	7	4	1	0	0	1	10	2	3	275
276	`082ac4e519b3d7f6`	104	12	8	2	42	11	5	9	1	0	0	1	10	2	3	276
277	`082b7c5a496e3f1d`	108	20	4	2	58	16	0	5	2	0	0	1	11	2	3	277
278	`08a75db391f6c4e2`	92	24	4	2	46	22	0	5	2	0	0	1	9	2	3	278
279	`086e4c295d7f1b3a`	90	20	6	2	42	9	15	0	3	0	0	1	9	2	3	279
280	`082a4ce7193bf5d6`	104	12	8	2	24	32	0	3	4	0	0	1	10	2	3	280
281	`082a4ce5193bd7f6`	104	12	8	2	48	8	8	3	4	0	0	1	10	2	3	281
282	`082ac4e319b7d5f6`	100	4	12	2	54	0	0	9	6	0	0	1	10	2	3	282
283	`086e4c295d7f3b1a`	62	36	2	2	36	21	12	0	0	1	0	1	8	2	3	283
284	`04ae8c239dbf5176`	62	36	2	2	36	21	12	0	0	1	0	1	8	2	3	284
285	`08a35df2c4e791b6`	60	32	4	2	32	27	0	7	0	1	0	1	7	2	3	285
286	`08e64c2bd5f7193a`	90	20	6	2	24	27	8	3	0	1	0	1	8	2	3	286
287	`0823d5fa4ce791b6`	88	16	8	2	38	13	8	4	2	1	0	1	8	2	3	287
288	`046153728ce9dbfa`	0	56	0	2	0	56	0	0	0	0	0	2	7	2	2	288
289	`046b59728ce3d1fa`	0	56	0	2	0	56	0	0	0	0	0	2	7	2	2	289
290	`0463d9f28ceb517a`	56	24	8	2	0	44	0	6	0	0	0	2	7	2	3	290
291	`0127456389aedcbf`	96	24	0	4	24	6	24	3	0	0	0	1	8	2	2	291
292	`081b2a394c5e7f6d`	64	32	0	4	0	36	0	12	0	0	0	1	6	2	2	292
293	`0c2f1d7b483a596e`	96	24	0	4	12	38	0	3	4	0	0	1	9	2	3	293
294	`082ac4e719b3d5f6`	88	8	8	4	12	20	0	9	4	2	0	1	7	2	3	294
295	`086e4c2b5d7f193a`	60	24	4	4	16	9	16	5	0	3	0	1	6	2	3	295
296	`082b5d7f193e4c6a`	84	0	12	4	36	3	0	0	12	3	0	1	7	2	3	296
297	`082b197e4c6f5d3a`	0	48	0	4	0	32	0	12	0	0	0	2	5	2	2	297
298	`046351728cebd9fa`	0	48	0	4	0	32	0	12	0	0	0	2	5	2	2	298
299	`082b5d7a4c6f193e`	56	16	8	4	0	26	0	12	0	2	0	2	5	2	3	299
300	`082a4c6f193b5d7e`	56	0	8	8	0	14	0	0	0	14	0	2	4	2	3	300
301	`082b193a4c6f5d7e`	0	32	0	8	0	0	0	24	0	0	0	4	3	2	2	301
302	`0123456789abcdef`	0	0	0	16	0	0	0	0	0	0	0	16	0	2	1	302