



COMPARATIVE EVALUATION OF OPEN-SOURCE IMPLEMENTATIONS OF GRÖBNER BASIS COMPUTATION ALGORITHMS FOR SMALL-SCALE AES POLYNOMIAL SYSTEMS OVER $\text{GF}(2)$

Bc. Pavel Holý

Master's thesis
Faculty of Information Technology
Czech Technical University in Prague
Department of Information Security
Study program: Informatics
Specialisation: Computer Security
Supervisor: Mgr. Martin Jureček, Ph.D.
May 24, 2026



Assignment of master's thesis

Title:	Comparative Evaluation of Open-Source Implementations of Gröbner Basis Computation Algorithms for Small-Scale AES Polynomial Systems over $\text{GF}(2)$
Student:	Bc. Pavel Holý
Supervisor:	Mgr. Martin Jureček, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Security
Department:	Department of Information Security
Validity:	until the end of summer semester 2026/2027

Instructions

This thesis focuses on the comparative evaluation of existing implementations of Gröbner basis computation based on the Faugère F4 algorithm for polynomial systems over the finite field $\text{GF}(2)$. Such polynomial systems naturally arise in the algebraic analysis of cryptographic primitives, including simplified variants of the AES cipher. Rather than developing a new Gröbner basis algorithm, the thesis aims to study, configure, and experimentally compare available implementations, with a particular emphasis on open-source tools. The evaluated systems will be applied to polynomial representations of small-scale AES variants, and their performance will be assessed in terms of computation time, memory consumption, and practical usability. The results will be analyzed and compared with the established reference system Magma, providing insights into the strengths and limitations of current open-source Gröbner basis solvers for cryptographic applications.

Instructions:

1. Study the theory of Gröbner bases and the main algorithms used for their computation, with an emphasis on F4-based methods.
2. Identify and analyze existing Gröbner basis implementations, with a focus on open-source tools supporting F4-based computation.
3. Model small-scale variants of the AES cipher as systems of multivariate polynomial equations over $\text{GF}(2)$.



4. Experimentally evaluate and compare selected Gröbner basis computation methods using AES-derived benchmark instances, measuring computation time and memory usage.
5. Attempt to improve an existing open-source implementation of the F4 algorithm, or alternatively implement a version of the F4 algorithm from scratch.
6. Analyze, compare, and critically discuss the experimental results, highlighting the practical strengths and limitations of open-source Gröbner basis solvers in cryptanalytic settings.



Czech Technical University in Prague

Faculty of Information Technology

© 2026 Bc. Pavel Holý. All rights reserved.

This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its use, with the exception of free statutory licenses and beyond the scope of the authorizations specified in the Declaration, requires the consent of the author.

Citation of this thesis: Holý Pavel. *Comparative Evaluation of Open-Source Implementations of Gröbner Basis Computation Algorithms for Small-Scale AES Polynomial Systems over $GF(2)$* . Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2026.

I would like to express my sincere gratitude to my supervisor, Mgr. Martin Jureček, Ph.D., for his guidance, patience, support, and valuable feedback during the preparation of this work. I would also like to thank my family and loved ones for their support. Finally, I would like to thank Ing. Jiří Buček, Ph.D., for helping me set up the software on the school server used for the experiments.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Czech Technical University in Prague has the right to conclude a licence agreement on the utilization of this thesis as a school work pursuant of Section 60 (1) of the Act.

In Prague on May 24, 2026

DECLARATION

I, the undersigned

Student's surname, given name(s): Holý Pavel
Personal number: 508699
Programme name: Informatics

hereby declare that the Master's Thesis titled

Comparative Evaluation of Open-Source Implementations of Gröbner Basis Computation Algorithms for Small-Scale AES Polynomial Systems over $GF(2)$

is my own independent work and that I have declared all sources of information used in accordance with the Methodological Guideline for adhering to ethical principles when elaborating an academic final thesis and the Framework Rules for the use of artificial intelligence at CTU for study and pedagogical purposes in Bachelor and continuing Masters studies.

I declare that I have used artificial intelligence tools during the preparation and writing of the final thesis.
I've verified the generated content. I confirm that I am aware that I am fully responsible for the content of the final thesis.

In Prague on 23.04.2026

Bc. Pavel Holý

.....
student's signature

Abstract

This thesis compares open-source implementations of Gröbner basis computation algorithms on polynomial systems derived from small-scale AES instances over $\text{GF}(2)$. The evaluation compares 20 open-source implementations with the commercial system Magma as a reference. The results show that performance strongly depends on the selected small-scale AES variant, the number of plaintext-ciphertext pairs, and the inclusion of Boolean polynomials. Adding Boolean polynomials improves performance in most cases. In the hardest experiment with Boolean polynomials, several open-source implementations successfully solve all instances, with `msolve` achieving the best overall performance. The thesis also proposes two implementation-level improvements to `msolve`, which reduce its runtime in the benchmarked setting.

Keywords Gröbner bases, F4 algorithm, algebraic cryptanalysis, AES, small-scale AES, $\text{GF}(2)$, Boolean polynomial rings, benchmark, open-source, `msolve`

Abstrakt

Tato práce se zabývá porovnáním open-source implementací algoritmů pro výpočet Gröbnerových bází na polynomiálních systémech odvozených ze zjednodušených variant AES nad $\text{GF}(2)$. Celkem je porovnáno 20 open-source implementací a komerční systém Magma, který slouží jako reference. Výsledky ukazují, že výkon jednotlivých implementací výrazně závisí na zvolené variantě AES, počtu dvojic otevřený text–šifrový text a na tom, zda jsou do systému přidány booleovské polynomy. Jejich přidání ve většině případů vede ke zlepšení výkonu. V nejtěžším experimentu s booleovskými polynomy dokázalo několik open-source implementací vyřešit všechny instance, přičemž nejlepšího celkového výsledku dosáhl msolve. Práce dále navrhuje dvě implementační úpravy systému msolve, které v měřeném prostředí zkracují jeho dobu běhu.

Klíčová slova Gröbnerovy báze, F4 algoritmus, algebraická kryptoanalýza, AES, zjednodušené varianty AES, $\text{GF}(2)$, booleovské polynomiální okruhy, benchmark, open-source, msolve

Contents

1	Introduction	1
2	Mathematical Background	3
2.1	Polynomial Systems over Finite Fields	3
2.1.1	Formal Polynomials and Polynomial Functions	4
2.2	Solution Sets and Ideals	5
2.3	Monomial Orderings and Leading Terms	6
2.3.1	Lexicographic Order	7
2.3.2	Graded Lexicographic Order	7
2.3.3	Graded Reverse Lexicographic Order	7
2.3.4	Leading Coefficient, Leading Monomial, and Leading Term	8
2.4	Polynomial Division and Normal Forms	8
2.5	Gröbner Bases	9
2.5.1	Reduced Gröbner Bases	10
2.5.2	Solving Polynomial Systems	10
2.6	S-Polynomials and Buchberger’s Criterion	10
2.7	Buchberger’s Algorithm	11
2.8	Matrix View of Polynomial Reduction	12
2.9	Faugère’s F4 Algorithm	12
2.9.1	Symbolic Preprocessing	13
2.10	Connection to the Benchmarks	14
3	Small-Scale AES	16
3.1	From AES to Small-Scale AES	16
3.2	Parameters of $SR(n,r,c,e)$	17
3.3	Encryption Process	18
3.3.1	SubBytes	18
3.3.2	ShiftRows	19
3.3.3	MixColumns	19
3.3.4	AddRoundKey and Key Schedule	20
3.4	Known Plaintexts and Polynomial Systems	21
3.5	Use in This Work	23
4	Evaluated Implementations	24
4.1	Singular	25
4.2	SageMath	27

4.3	BRiAl	29
4.4	CoCoALib	30
4.5	Giac	31
4.6	msolve	33
4.7	Macaulay2	35
4.8	M4GB	37
4.9	Groebner.jl	39
4.10	openf4	40
4.11	Magma	41
5	Benchmarking Setup	44
5.1	Architecture Overview	44
5.2	Input Generator	46
5.3	Benchmark Runner	49
5.3.1	Benchmark Suite Specification	49
5.3.2	Unified Interface Layer	51
5.3.2.1	The PRE Interface	51
5.3.2.2	The MAIN Interface	52
5.3.2.3	The POST Interface	52
5.3.3	Lifecycle of a Benchmark Run	53
5.3.4	Resource Control Using cgroups v2 and <code>systemd-run</code>	54
5.3.4.1	Approach 1 – <code>time -v</code>	57
5.3.4.2	Approach 2 – Process Groups	57
5.3.4.3	Approach 3 – Traversing the Process Tree	58
5.3.4.4	The Final Approach – cgroups v2 and <code>systemd-run</code>	59
5.3.5	Parallel Execution and Scheduling	63
5.3.6	Invocation and CLI	64
5.4	Output Parser	66
6	Experimental Results	68
6.1	Experiments Without Boolean Polynomials	70
6.1.1	Pilot Experiment Without Boolean Polynomials	71
6.1.2	Easy Experiment Without Boolean Polynomials	73
6.1.3	Medium Experiment Without Boolean Polynomials	75
6.1.4	Hard Experiment Without Boolean Polynomials	77
6.2	Experiments With Boolean Polynomials	78
6.2.1	Pilot Experiment With Boolean Polynomials	78
6.2.2	Easy Experiment With Boolean Polynomials	79
6.2.3	Medium Experiment With Boolean Polynomials	82
6.2.4	Hard Experiment With Boolean Polynomials	83
6.3	Summary	85

7	Improving msolve	87
7.1	Caching Leading Monomial Metadata	87
7.2	Using Cached Degrees for Reducer Filtering	91
7.3	Results	92
8	Conclusion	96
	Bibliography	98
	Contents of attached files	108

List of Figures

3.1	State numbering used by small-scale AES	18
3.2	High-level encryption structure of $SR(n,r,c,e)$	18
3.3	AES SubBytes operation	19
3.4	AES ShiftRows operation	20
3.5	AES MixColumns operation	20
3.6	AES AddRoundKey operation	21
3.7	AES key schedule	22
5.1	High-level overview of the benchmarking framework	45
5.2	Overview of a benchmark run	56
6.1	Runtime of the Pilot experiment without Boolean polynomials	72
6.2	Peak memory consumption of the Pilot experiment without Boolean polynomials	72
6.3	Runtime of the Easy experiment without Boolean polynomials	73
6.4	Peak memory consumption of the Easy experiment without Boolean polynomials	74
6.5	Runtime of the Medium experiment without Boolean polynomials	76
6.6	Peak memory consumption of the Medium experiment without Boolean polynomials	76
6.7	Runtime of the Hard experiment without Boolean polynomials	77
6.8	Peak memory consumption of the Hard experiment without Boolean polynomials	78
6.9	Runtime of the Pilot experiment with Boolean polynomials . .	79
6.10	Peak memory consumption of the Pilot experiment with Boolean polynomials	80
6.11	Runtime of the Easy experiment with Boolean polynomials . .	81
6.12	Peak memory consumption of the Easy experiment with Boolean polynomials	81
6.13	Runtime of the Medium experiment with Boolean polynomials	83
6.14	Peak memory consumption of the Medium experiment with Boolean polynomials	83
6.15	Runtime of the Hard experiment with Boolean polynomials . .	84
6.16	Peak memory consumption of the Hard experiment with Boolean polynomials	85
7.1	msolve runtime improvement comparison	93

7.2	msolve peak memory consumption improvement comparison . .	94
7.3	msolve CPU usage improvement comparison	94

List of Tables

3.1	Parameters of $SR(n,r,c,e)$	17
5.1	Test of <code>time -v</code> on two processes	57
6.1	Summary of the Pilot experiment without Boolean polynomials	71
6.2	Summary of the Easy experiment without Boolean polynomials	74
6.3	Summary of the Medium experiment without Boolean polynomials	75
6.4	Summary of the Hard experiment without Boolean polynomials	77
6.5	Summary of the Pilot experiment with Boolean polynomials . .	79
6.6	Summary of the Easy experiment with Boolean polynomials . .	80
6.7	Summary of the Medium experiment with Boolean polynomials	82
6.8	Summary of the Hard experiment with Boolean polynomials . .	84
7.1	msolve runtime improvement comparison	92
7.2	msolve peak memory consumption improvement comparison . .	93
7.3	msolve CPU usage improvement comparison	93
7.4	msolve modification comparison on two systems	95

List of Code listings

4.1	Singular input example	26
4.2	Singular output example	26
4.3	Sage script example	28
4.4	Sage output example of <code>toy:buchberger</code>	28
4.5	Sage output example of <code>toy:buchberger2</code> and <code>toy:d_basis</code> . . .	28
4.6	BRiAl source code example	31
4.7	BRiAl output example	31
4.8	CoCoALib source code example	32
4.9	CoCoALib output example	32
4.10	Giac input example	33

4.11	Giac output example	34
4.12	msolve input example	35
4.13	msolve output example	35
4.14	Macaulay2 input example over polynomial ring	36
4.15	Macaulay2 output example over polynomial ring	36
4.16	Macaulay2 input example over Boolean polynomial ring	37
4.17	Macaulay2 output example over Boolean polynomial ring	37
4.18	M4GB input example	38
4.19	M4GB output example	38
4.20	Groebner.jl input example	40
4.21	Groebner.jl output example	40
4.22	openf4 source code example	42
4.23	openf4 output example	42
4.24	Magma input example	43
4.25	Magma output example	43
5.1	<code>inputs-spec.json</code> example	47
5.2	Generated <code>.in</code> file example	48
5.3	<code>benchmarks-spec.json</code> example	50
5.4	Example of <code>result.txt</code>	53
5.5	<code>run.log</code> example	55
5.6	<code>systemd-run</code> invocation	60
5.7	Configuring a <code>systemd</code> slice	62
5.8	Generated JSON from Output parser	67
6.1	Pilot experiment input specification without Boolean polynomials	69
6.2	Pilot experiment benchmark specification without Boolean polynomials	70
7.1	Caching leading monomial exponent vectors and degrees	89
7.2	Passing cached leading monomial metadata to reducer lookup	89
7.3	Extended reducer lookup signature	90
7.4	Using the cached exponent vector of a leading monomial	90
7.5	Degree-based filtering of reducer candidates	91
7.6	Exact component-wise monomial divisibility check	92

List of Algorithms

2.1	Buchberger's Algorithm	11
2.2	F4 Algorithm	13
2.3	ComputeM	14

List of abbreviations

BFS	breadth-first search
CLI	command-line interface
CPU	central processing unit
CT	ciphertext
DAG	directed acyclic graph
FD	file descriptor
GF	Galois field
GPU	graphics processing unit
I/O	input/output
L3	level 3 cache
LC	leading coefficient
LM	leading monomial
LT	leading term
NP	nondeterministic polynomial time
PID	process identifier
PT	plaintext
PT-CT	plaintext-ciphertext
REPL	read-eval-print loop
STDERR	standard error stream
STDOUT	standard output stream
UID	user identifier
UUID4	universally unique identifier, version 4

[illegible]

Introduction

As more and more aspects of everyday life are being moved to the digital world, cryptosystems are becoming more important than ever. At the same time, the incentive for malicious actors to break these systems increases as well. It is therefore crucial that the cryptosystems we use are robust and resilient to various attacks.

AES [1] is one of the most commonly used ciphers today. The length of its key makes it infeasible to break by brute force. It has also withstood standard techniques such as linear and differential cryptanalysis. In this work, we therefore focus on algebraic cryptanalysis, which consists of two main steps. First, the cryptosystem is converted into a polynomial system over a finite field. Then, the secret key is recovered by solving this system using existing methods, such as SAT solvers or Gröbner basis computation. Deciding whether a polynomial system has a solution is an NP-complete problem, even if all equations are quadratic and the field is $\text{GF}(2)$ [2]. To the authors' knowledge, Faugère's F4 algorithm is one of the fastest algorithms for Gröbner basis computation.

For a fixed key, AES can be viewed as a map from a finite set of plaintexts to a finite set of ciphertexts. In the known-plaintext attack setting considered in this work, the key is fixed but unknown, and the observed plaintext-ciphertext pairs impose constraints on the same key bits. By the Universal Mapping Theorem, these constraints can be represented by polynomial equations over $\text{GF}(p)$ for some prime number p , where the unknown variables correspond to the key bits. We focus on the simplest case, $\text{GF}(2)$.

Each plaintext-ciphertext pair can be converted into a polynomial system with the number of equations equal to the number of key bits. By using more than one such pair, an overdefined system is obtained. This can lead to faster runtimes for some solvers. [3]

Using algebraic cryptanalysis to attack the full AES cipher with currently available methods is infeasible, as the number of constructed polynomials and their complexity are too large. We therefore focus on small-scale AES variants.

These are simplified variants of the AES cipher that reduce the number of rounds and the size of the internal state, while preserving an algebraic structure similar to that of AES [4]. As a result, the variants selected for this work lead to polynomial systems that are solvable with current methods.

The goal of this work is to provide an experimental evaluation of open-source implementations for Gröbner basis computation in the context of algebraic cryptanalysis. We focus on polynomial systems derived from small-scale AES variants over $\text{GF}(2)$ and compare a broad selection of available implementations with respect to runtime, memory usage, success rate, and other metrics. The results are compared with the commercial system Magma [5], providing a discussion of the current state of open-source implementations. To support this evaluation, we developed a custom benchmarking framework that generates input systems, executes implementations under controlled resource limits, and collects performance metrics.

Finally, we attempt to improve `msolve` [6], which is one of the open-source implementations. We propose two modifications that lead to substantial runtime improvements.

The rest of this work is organized as follows. Chapter 2 introduces the theory behind Gröbner bases and outlines the F4 algorithm. Chapter 3 provides an overview of the small-scale AES variants, which are used in the following experiments. Chapter 4 lists the implementations we evaluate, together with their properties and usage examples. Chapter 5 describes the benchmarking framework we created in detail. Chapter 6 presents and discusses the experimental results, highlighting the strengths and weaknesses of the implementations. Chapter 7 proposes two improvements to the `msolve` implementation and measures their performance. Finally, Chapter 8 concludes this work and outlines directions for future research.

Mathematical Background

This chapter introduces the mathematical background needed in the rest of the work. We focus only on the concepts that are used later: polynomial systems over finite fields, ideals, monomial orderings, polynomial division, Gröbner bases, Buchberger’s algorithm, and Faugère’s F4 algorithm. Throughout this chapter, we use the notation from [7].

The goal of the chapter is not to provide a complete introduction of computational algebra. We only need enough theory to explain what the evaluated implementations compute and why these computations can be used for solving polynomial systems derived from small-scale AES.

2.1 Polynomial Systems over Finite Fields

Let k be a field. Informally, a field is a set in which addition, subtraction, multiplication, and division by nonzero elements are defined and behave as expected. Standard examples are $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$. In this work, however, we mainly work over the field with two elements. It is commonly denoted by $\mathbb{F}_2$ or $\text{GF}(2)$. We introduce both notations here, but in the rest of the work we use $\text{GF}(2)$. [7, pp. 1, 5]

The field $\text{GF}(2)$ consists of the elements 0 and 1, with arithmetic modulo two. In particular, $1+1=0$. Addition in $\text{GF}(2)$ therefore corresponds to XOR on bits, and subtraction is the same operation as addition. Multiplication corresponds to logical AND on individual bits. Since the polynomial systems used in this work describe bit-oriented ciphers, $\text{GF}(2)$ is the natural coefficient field. [7, p. 5]

A monomial in variables $x_1, \dots, x_n$ is a product

$$x_1^{\alpha_1} x_2^{\alpha_2} \cdots x_n^{\alpha_n},$$

where all exponents α_i are nonnegative integers. If $\alpha = (\alpha_1, \dots, \alpha_n) \in \mathbb{Z}_{\geq 0}^n$, we write

$$x^\alpha = x_1^{\alpha_1} \cdots x_n^{\alpha_n}.$$

The total degree of x^α is $|\alpha| = \alpha_1 + \dots + \alpha_n$. If $\alpha = (0, \dots, 0)$, then $x^\alpha = 1$. [7, p. 2]

A polynomial over k is a finite linear combination of monomials,

$$f = \sum_{\alpha} a_{\alpha} x^{\alpha}, \quad a_{\alpha} \in k,$$

where only finitely many coefficients a_{α} are nonzero. The set of all polynomials in $x_1, \dots, x_n$ with coefficients in k is denoted by $k[x_1, \dots, x_n]$ and is called a polynomial ring. For a nonzero polynomial f , the total degree $\deg(f)$ is the maximum of $|\alpha|$ over all monomials with nonzero coefficient in f . The degree of the zero polynomial is undefined. [7, pp. 2–3]

In this work, the typical polynomial ring is

$$R = \text{GF}(2)[x_1, \dots, x_n].$$

A polynomial system is a finite set or tuple

$$F = (f_1, \dots, f_s), \quad f_i \in R,$$

which we interpret as the system of equations

$$f_1 = 0, \quad \dots, \quad f_s = 0.$$

Later, the variables represent unknown key bits, and the equations are derived from known plaintext-ciphertext pairs of small-scale AES instances.

2.1.1 Formal Polynomials and Polynomial Functions

A polynomial can be viewed in two related ways. First, it is a formal object in the polynomial ring $k[x_1, \dots, x_n]$. In this view, two polynomials are equal when all coefficients of corresponding monomials are equal. Second, a polynomial can be evaluated at points of affine space and therefore defines a function

$$f : k^n \rightarrow k, \quad (a_1, \dots, a_n) \mapsto f(a_1, \dots, a_n).$$

Thus, the statement $f = 0$ can mean that f is the zero polynomial, or it can mean that f is the zero function (that is, $f(a) = 0$ for all $a \in k^n$). For infinite fields these two meanings are equivalent, but for finite fields they can differ. [7, pp. 3–4]

This distinction matters over $\text{GF}(2)$. Consider the polynomial

$$x^2 - x \in \text{GF}(2)[x].$$

Since subtraction and addition are the same in $\text{GF}(2)$, this is the same formal polynomial as $x^2 + x$. It is not the zero polynomial, because the monomials x^2

and x both have coefficient 1. However, as a function on $\text{GF}(2)$, it evaluates to zero at both possible inputs:

$$0^2 - 0 = 0, \quad 1^2 - 1 = 0.$$

Therefore, $x^2 - x$ is a nonzero formal polynomial that induces the zero function on $\text{GF}(2)$. [7, p. 3]

Gröbner basis algorithms work with formal polynomials. They do not automatically replace x_i^2 by x_i just because the intended value of x_i is a bit. We must encode this fact explicitly. One option is to work in the ordinary polynomial ring $\text{GF}(2)[x_1, \dots, x_n]$ and add the Boolean equations

$$x_i^2 - x_i = 0, \quad 1 \leq i \leq n.$$

Over $\text{GF}(2)$, the same equations can be written as $x_i^2 + x_i = 0$. Another option is to work in the quotient ring

$$\text{GF}(2)[x_1, \dots, x_n] / \langle x_1^2 - x_1, \dots, x_n^2 - x_n \rangle.$$

In this ring, every variable satisfies $x_i^2 = x_i$, and monomials can be reduced to square-free monomials. Quotient rings are introduced in the general form $k[x_1, \dots, x_n]/I$ in [7, pp. 240–246].

This is also a practical distinction. Some implementations evaluated in this work use ordinary polynomial rings and require the Boolean equations as part of the input. Other implementations support Boolean polynomial rings or quotient rings directly. This affects the input format.

2.2 Solution Sets and Ideals

For a field k , the n -dimensional affine space over k is

$$k^n = \{(a_1, \dots, a_n) \mid a_i \in k\}.$$

For polynomials $f_1, \dots, f_s \in k[x_1, \dots, x_n]$, the affine variety defined by them is

$$V(f_1, \dots, f_s) = \{(a_1, \dots, a_n) \in k^n \mid f_i(a_1, \dots, a_n) = 0 \text{ for all } 1 \leq i \leq s\}.$$

In this work, we mostly use the term solution set, because it describes the role of this object in cryptanalysis more directly. The notation $V(f_1, \dots, f_s)$ is still useful. [7, pp. 5–6]

An ideal I of the polynomial ring $k[x_1, \dots, x_n]$ is a subset satisfying three properties:

$$0 \in I,$$

$$f, g \in I \implies f + g \in I,$$

$$f \in I, h \in k[x_1, \dots, x_n] \implies hf \in I.$$

If $f_1, \dots, f_s$ are polynomials, the ideal generated by them is

$$\langle f_1, \dots, f_s \rangle = \left\{ \sum_{i=1}^s h_i f_i \mid h_i \in k[x_1, \dots, x_n] \right\}.$$

We can view this ideal as the set of all “polynomial consequences” obtained from the equations $f_1 = 0, \dots, f_s = 0$ by multiplying equations by polynomials and adding the results. [7, pp. 29–30]

For a system $F = (f_1, \dots, f_s)$, we write

$$\langle F \rangle = \langle f_1, \dots, f_s \rangle.$$

Different generating sets may generate the same ideal. If

$$\langle f_1, \dots, f_s \rangle = \langle g_1, \dots, g_t \rangle,$$

then the two systems have the same solution set,

$$V(f_1, \dots, f_s) = V(g_1, \dots, g_t).$$

This is the reason why we can replace the original equations by a Gröbner basis: the equations may change, but the generated ideal, and hence the solution set, remains the same. [7, p. 31]

If $1 \in \langle F \rangle$, then the generated ideal is the whole polynomial ring. In this case, the system has no solution, because every solution would have to satisfy the equation $1 = 0$.

2.3 Monomial Orderings and Leading Terms

For polynomials in one variable, there is a natural order of monomials:

$$1 < x < x^2 < x^3 < \dots.$$

For several variables, we must choose how to compare monomials such as x^2y and xy^2 . This choice is made by a monomial ordering. It determines leading terms, polynomial division, Gröbner bases, and also the ordering of columns in F4 matrices. [7, pp. 54–55]

A monomial ordering is an ordering $>$ on exponent vectors in $\mathbb{Z}_{\geq 0}^n$, equivalently on monomials x^α , with three properties. First, it is a total order. This means that any two monomials can be compared: for every pair of monomials x^α and x^β , exactly one of the relations

$$x^\alpha > x^\beta, \quad x^\alpha = x^\beta, \quad x^\beta > x^\alpha$$

holds. The order is also transitive. Second, it is compatible with multiplication by monomials, which means that

$$\alpha > \beta \implies \alpha + \gamma > \beta + \gamma$$

for all $\alpha, \beta, \gamma \in \mathbb{Z}_{\geq 0}^n$. Finally, it is a well-ordering, meaning that every nonempty set of monomials has a smallest element. [7, pp. 55–56]

The well-ordering property is important for termination. During polynomial reduction, the current leading monomial is repeatedly replaced by a smaller one. If this could continue forever, we would obtain an infinite strictly decreasing sequence of monomials. A monomial ordering does not allow such a sequence. This is the same argument used when proving termination of multivariate division and the symbolic preprocessing step of F4. [7, p. 56]

2.3.1 Lexicographic Order

Assume the variables are ordered as $x_1 > x_2 > \dots > x_n$. The lexicographic ordering compares exponent vectors from left to right. We write $\alpha >_{\text{lex}} \beta$ if the first nonzero entry of $\alpha - \beta$ is positive. This ordering is useful for elimination and for obtaining a triangular-like structure of equations. However, computing a Gröbner basis directly with lexicographic order is often expensive. [7, pp. 56–57, 97–100]

2.3.2 Graded Lexicographic Order

The graded lexicographic ordering first compares total degrees. We write $\alpha >_{\text{grlex}} \beta$ if either $|\alpha| > |\beta|$, or $|\alpha| = |\beta|$ and $\alpha >_{\text{lex}} \beta$. Thus total degree is the primary criterion, and lexicographic order is used only to break ties. [7, p. 58]

2.3.3 Graded Reverse Lexicographic Order

The graded reverse lexicographic ordering also compares total degrees first. If $|\alpha| = |\beta|$, the comparison is decided from the right. We write $\alpha >_{\text{grevlex}} \beta$ if the rightmost nonzero entry of $\alpha - \beta$ is negative. [7, pp. 58–59]

This ordering is important in practice. Lexicographic order is useful when we want equations suitable for elimination, but efficient Gröbner basis computations often use grevlex-like orderings for the main computation. This is one reason why many implementations and benchmark configurations prefer grevlex when it is available. [7, p. 119]

2.3.4 Leading Coefficient, Leading Monomial, and Leading Term

Fix a monomial ordering $>$. For a nonzero polynomial

$$f = \sum_{\alpha} a_{\alpha} x^{\alpha},$$

we define the multidegree as

$$\text{multideg}(f) = \max\{\alpha \mid a_{\alpha} \neq 0\},$$

where the maximum is taken with respect to the chosen ordering. The leading coefficient, leading monomial, and leading term are

$$\text{LC}(f) = a_{\text{multideg}(f)}, \quad \text{LM}(f) = x^{\text{multideg}(f)}, \quad \text{LT}(f) = \text{LC}(f) \text{LM}(f).$$

All three depend on the monomial ordering. [7, pp. 59–60]

Over $\text{GF}(2)$, every nonzero coefficient is 1. Therefore, for nonzero polynomials over $\text{GF}(2)$, the distinction between $\text{LT}(f)$ and $\text{LM}(f)$ is often only notational. We still keep the standard notation because the algorithms are usually stated over arbitrary fields.

2.4 Polynomial Division and Normal Forms

Multivariate polynomial division generalizes ordinary polynomial long division. Given a polynomial f and an ordered tuple $F = (f_1, \dots, f_s)$, the division algorithm produces polynomials $q_1, \dots, q_s$ and a remainder r such that

$$f = q_1 f_1 + \dots + q_s f_s + r.$$

The remainder is either zero or a linear combination of terms that are not divisible by any of $\text{LT}(f_1), \dots, \text{LT}(f_s)$. Moreover, if $q_i f_i \neq 0$, then the multidegree of $q_i f_i$ is not larger than the multidegree of f . [7, p. 64]

The algorithm maintains an intermediate polynomial p , initially equal to f . At each step, it checks whether the current leading term $\text{LT}(p)$ is divisible by some $\text{LT}(f_i)$. If such a divisor exists, the algorithm subtracts a suitable multiple of f_i from p so that the current leading term is canceled. If no such divisor exists, $\text{LT}(p)$ is moved to the remainder. The algorithm terminates because the leading term of p strictly decreases, and the well-ordering property prevents an infinite strictly decreasing sequence of monomials. [7, pp. 65–66]

For an arbitrary ordered tuple F , the remainder is not uniquely determined by the ideal. It can depend on the order of $f_1, \dots, f_s$, and it can also change when the monomial ordering changes. [7, pp. 67–68]

This changes when we divide by a Gröbner basis. If G is a Gröbner basis, the remainder of f on division by G is unique. We denote it by

$$\bar{f}^G.$$

If G is a Gröbner basis of an ideal I , then

$$f \in I \iff \bar{f}^G = 0.$$

This gives an effective ideal membership test. [7, pp. 83–84]

2.5 Gröbner Bases

Let $I \subseteq k[x_1, \dots, x_n]$ be an ideal. The set of leading terms of nonzero elements of I is

$$\text{LT}(I) = \{\text{LT}(f) \mid f \in I, f \neq 0\}.$$

The ideal generated by these leading terms is denoted by $\langle \text{LT}(I) \rangle$. If I is generated by $f_1, \dots, f_s$, the ideal generated only by the leading terms of these chosen generators,

$$\langle \text{LT}(f_1), \dots, \text{LT}(f_s) \rangle,$$

may miss leading terms of other polynomials in I . In other words, the chosen generators do not necessarily generate the full leading-term ideal. Gröbner bases are generating sets that avoid this problem. [7, pp. 76–78]

A finite subset $G = \{g_1, \dots, g_t\} \subseteq I$ is a Gröbner basis of I if

$$\langle \text{LT}(g_1), \dots, \text{LT}(g_t) \rangle = \langle \text{LT}(I) \rangle.$$

This is the formal definition. Informally, the leading terms of the elements of G generate all leading terms that are in the ideal I . [7, p. 78]

There is also a useful divisibility interpretation. A finite subset $G \subseteq I$ is a Gröbner basis if and only if, for every nonzero polynomial $f \in I$, the leading term $\text{LT}(f)$ is divisible by at least one of the leading terms $\text{LT}(g_i)$. [7, p. 78]

A Gröbner basis is still a generating set of the original ideal. Therefore, if $I = \langle f_1, \dots, f_s \rangle$ and $G = \{g_1, \dots, g_t\}$ is a Gröbner basis of I , then

$$\langle g_1, \dots, g_t \rangle = I$$

and

$$V(f_1, \dots, f_s) = V(g_1, \dots, g_t).$$

The system $g_1 = 0, \dots, g_t = 0$ can therefore replace the original system $f_1 = 0, \dots, f_s = 0$ without changing the set of solutions. [7, pp. 31, 77–78]

Every nonzero ideal in $k[x_1, \dots, x_n]$ has a finite Gröbner basis. This follows from the Hilbert Basis Theorem. The result is important because the algorithms below construct a finite object whose existence is guaranteed. [7, pp. 76–80]

2.5.1 Reduced Gröbner Bases

A Gröbner basis G is reduced if every polynomial $p \in G$ has leading coefficient 1, and no monomial of p lies in the ideal generated by the leading terms of $G \setminus \{p\}$. For a fixed ideal and a fixed monomial ordering, the reduced Gröbner basis is unique. [7, pp. 93–94]

This uniqueness is useful for comparing outputs. Two implementations may compute different non-reduced Gröbner bases for the same input, and both outputs can be correct. A reduced Gröbner basis gives a canonical target once the field, ring, variable order, and monomial order are fixed.

2.5.2 Solving Polynomial Systems

Gröbner bases are useful for solving polynomial systems because they preserve the solution set while often changing the equations into a more useful form. Lexicographic order is especially useful for this purpose. A lexicographic Gröbner basis may contain equations involving fewer variables, which allows us to solve the system by solving one equation and substituting the result into the next ones. [7, pp. 97–100, 121–122]

For cryptanalytic systems, computing directly with lexicographic order is often too expensive. A common strategy is to compute a Gröbner basis using a more efficient ordering, such as grevlex, and only later convert the basis if a lexicographic form is needed. In this work, we measure the cost of the Gröbner basis computation itself.

2.6 S-Polynomials and Buchberger's Criterion

The main difficulty in constructing a Gröbner basis is that combinations of generators can cancel leading terms and produce new leading terms that are not visible in the original generating set. S-polynomials isolate this situation.

Let $f, g \in k[x_1, \dots, x_n]$ be nonzero polynomials. If $\text{LM}(f) = x^\alpha$ and $\text{LM}(g) = x^\beta$, then

$$\text{lcm}(\text{LM}(f), \text{LM}(g)) = x^\gamma, \quad \gamma_i = \max(\alpha_i, \beta_i).$$

The S-polynomial of f and g is

$$S(f, g) = \frac{x^\gamma}{\text{LT}(f)}f - \frac{x^\gamma}{\text{LT}(g)}g.$$

The two multiples have the same leading term, and the subtraction cancels it. [7, pp. 84–85]

Buchberger's criterion states that a basis $G = \{g_1, \dots, g_t\}$ of an ideal I is a Gröbner basis if and only if every S-polynomial reduces to zero modulo G :

$$\overline{S(g_i, g_j)}^G = 0 \quad \text{for all } i \neq j.$$

This criterion turns the definition of a Gröbner basis into an algorithmic test. [7, p. 86]

If some S-polynomial has a nonzero remainder, this remainder is a new polynomial consequence of the original system. Adding it to the current generating set brings us closer to a Gröbner basis. This is the basic idea behind Buchberger's algorithm.

2.7 Buchberger's Algorithm

Buchberger's algorithm is the classical constructive algorithm for computing Gröbner bases. The algorithm starts with the input polynomials. It repeatedly processes critical pairs, meaning unordered pairs $\{p, q\}$ of polynomials from the current basis. Each critical pair determines the corresponding S-polynomial $S(p, q)$, which is then reduced by the current basis. If the remainder is nonzero, it is added to the basis. The process stops when no critical pair produces a new polynomial. We present the basic version in Algorithm 2.1. [7, pp. 90–92]

Algorithm 2.1 Buchberger's Algorithm

Input: $F = \{f_1, \dots, f_s\}$, a finite set of polynomials

Output: A Gröbner basis $G = \{g_1, \dots, g_t\}$ of the ideal generated by F , with $F \subseteq G$

```

1:  $G \leftarrow F$ 
2: repeat
3:    $G' \leftarrow G$ 
4:   for all pairs  $\{p, q\}$  with  $p \neq q$  in  $G'$  do
5:      $r \leftarrow \overline{S(p, q)}^{G'}$ 
6:     if  $r \neq 0$  then
7:        $G \leftarrow G \cup \{r\}$ 
8:     end if
9:   end for
10: until  $G = G'$ 
11: return  $G$ 
    
```

The correctness argument follows the structure of the algorithm. Every polynomial added to G is obtained by reducing an S-polynomial of elements already in the ideal generated by F . Therefore, the generated ideal does not change. When the algorithm stops, all S-polynomials reduce to zero with respect to the current set G . By Buchberger's criterion, this set is a Gröbner basis. Termination follows because every nonzero remainder adds new leading-term information, and the corresponding ascending chains of leading-term ideals stabilize. [7, pp. 91–92]

The basic algorithm is simple, but it is inefficient on larger inputs. It can process many critical pairs whose S-polynomials reduce to zero. It also

reduces S-polynomials one by one, and intermediate polynomials can grow quickly. Several refinements that reduce this overhead are explained in [7, pp. 109–116].

2.8 Matrix View of Polynomial Reduction

The F4 algorithm is based on the observation that polynomial reductions can be represented by linear algebra. Let H be a finite set of polynomials. We denote by $\text{Mon}(H)$ the set of all monomials appearing in the polynomials of H . If we sort the monomials in $\text{Mon}(H)$ in decreasing order and use them as matrix columns, then each polynomial in H can be written as one row of coefficients. This is the matrix representation used in F4. [7, p. 571]

For example, let

$$H = \{x + y + 1, xy + y\} \subseteq \text{GF}(2)[x, y]$$

and let the columns be ordered as $xy > x > y > 1$. Then the coefficient rows are

$$x + y + 1 \mapsto (0, 1, 1, 1),$$

$$xy + y \mapsto (1, 0, 1, 0).$$

Over $\text{GF}(2)$, all entries are 0 or 1, row addition is XOR, and there is no nontrivial row scaling because the only nonzero scalar is 1.

Row operations correspond to linear combinations of polynomials. Eliminating a pivot column corresponds to canceling a leading monomial. Thus a row reduction can perform many cancellations at once. This is the main computational idea behind F4: instead of reducing S-polynomials individually by repeated division, the algorithm collects many polynomial multiples into a matrix and reduces them simultaneously. [7, pp. 567–568]

2.9 Faugère’s F4 Algorithm

Faugère’s F4 algorithm builds on Buchberger’s algorithm. The main bottleneck in Buchberger’s algorithm is the repeated reduction of S-polynomials. F4 reduces this cost by processing selected critical pairs in batches and by replacing many polynomial reductions with one matrix row reduction. We use the notation from [7, pp. 567–575].

The algorithm maintains a current basis G and a set B of critical pairs. In each iteration, it selects a nonempty subset $B' \subseteq B$. For every selected pair, it constructs the two polynomial multiples that form the two parts of the S-polynomial. These multiples are collected in a set L . The procedure `COMPUTEM`, described below, adds further multiples of polynomials from the current basis G whose leading monomials can reduce monomials already appearing in the constructed set. It then returns the corresponding coefficient

matrix. After row reduction, rows that produce new leading monomials are converted back into polynomials and added to G . We present the algorithm in Algorithm 2.2. [7, pp. 568–569]

Algorithm 2.2 F4 Algorithm

Input: $F = \{f_1, \dots, f_s\}$, a finite set of polynomials

Output: A Gröbner basis $G = \{g_1, \dots, g_t\}$ of the ideal generated by F , with $F \subseteq G$

```

1:  $G \leftarrow F$ 
2:  $t \leftarrow s$ 
3:  $B \leftarrow \{\{i, j\} \mid 1 \leq i < j \leq s\}$ 
4: while  $B \neq \emptyset$  do
5:   Select  $B' \subseteq B$ ,  $B' \neq \emptyset$ 
6:    $B \leftarrow B \setminus B'$ 
7:    $L \leftarrow \bigcup_{\{i,j\} \in B'} \left\{ \frac{\text{lcm}(\text{LM}(f_i), \text{LM}(f_j))}{\text{LT}(f_i)} \cdot f_i, \frac{\text{lcm}(\text{LM}(f_i), \text{LM}(f_j))}{\text{LT}(f_j)} \cdot f_j \right\}$ 
8:    $M \leftarrow \text{COMPUTEM}(L, G)$ 
9:    $N \leftarrow \text{row-reduced echelon form of } M$ 
10:   $N^+ \leftarrow \{n \in \text{rows}(N) \mid \text{LM}(n) \notin \langle \text{LM}(\text{rows}(M)) \rangle\}$ 
11:  for all  $n \in N^+$  do
12:     $t \leftarrow t + 1$ 
13:     $f_t \leftarrow \text{polynomial form of } n$ 
14:     $G \leftarrow G \cup \{f_t\}$ 
15:     $B \leftarrow B \cup \{\{i, t\} \mid 1 \leq i < t\}$ 
16:  end for
17: end while
18: return  $G$ 

```

The set B' determines which critical pairs are processed together. A common choice is the normal selection strategy. For a pair $\{i, j\}$, we define

$$\deg(\{i, j\}) = \deg(\text{lcm}(\text{LM}(f_i), \text{LM}(f_j))).$$

The normal selection strategy chooses the pairs with minimal value of this degree. This tends to process low-degree consequences first and helps control the size of the matrices. [7, pp. 569–570]

The set L contains the two multiples for each selected pair. Unlike Buchberger's algorithm, F4 does not have to form and reduce each S-polynomial separately. The necessary cancellations are produced later by row reduction of the matrix returned by COMPUTEM.

2.9.1 Symbolic Preprocessing

The matrix used by F4 must contain enough rows to simulate polynomial reduction. This is the role of COMPUTEM, also called symbolic preprocessing.

The procedure starts from the set L . It then inspects monomials that appear in the current set of rows. Whenever a monomial can be reduced by a leading monomial of some polynomial in G , the corresponding multiple of that polynomial is added as another row. We present the procedure in Algorithm 2.3. [7, pp. 571–572]

Algorithm 2.3 ComputeM

Input: $L, G = \{f_1, \dots, f_t\}$

Output: M

```

1:  $H \leftarrow L$ 
2:  $done \leftarrow \text{LM}(H)$ 
3: while  $done \neq \text{Mon}(H)$  do
4:   Select largest  $x^\beta \in \text{Mon}(H) \setminus done$  with respect to  $>$ 
5:    $done \leftarrow done \cup \{x^\beta\}$ 
6:   if there exists  $f_\ell \in G$  such that  $\text{LM}(f_\ell) \mid x^\beta$  then
7:     Select any  $f_\ell \in G$  such that  $\text{LM}(f_\ell) \mid x^\beta$ 
8:      $H \leftarrow H \cup \left\{ \frac{x^\beta}{\text{LM}(f_\ell)} \cdot f_\ell \right\}$ 
9:   end if
10: end while
11:  $M \leftarrow$  matrix of coefficients of  $H$  with respect to  $\text{Mon}(H)$ , columns in
    decreasing order according to  $>$ 
12: return  $M$ 
    
```

The procedure terminates for the same reason as polynomial division. Each processed monomial is marked as done. Any new monomials introduced by an added reducer multiple are smaller than the monomial currently being processed. Therefore, the loop cannot create an infinite descending sequence of monomials. [7, pp. 571–572]

After symbolic preprocessing, F4 reduces the matrix M . The rows of the reduced matrix are interpreted again as polynomials. Rows in N^+ correspond to new polynomial consequences with leading monomials not already covered by the rows used to construct M . These polynomials are added to G , and new critical pairs are created. In this way, F4 obtains the same kind of information as nonzero S-polynomial remainders in Buchberger’s algorithm, but it obtains them in batches by linear algebra. [7, pp. 568–570]

2.10 Connection to the Benchmarks

The benchmark instances used later follow the same algebraic workflow. A small-scale AES instance and one or more plaintext-ciphertext pairs are converted into a polynomial system

$$F_{\text{AES}} \subseteq \text{GF}(2)[x_1, \dots, x_n].$$

This system generates the ideal

$$I_{\text{AES}} = \langle F_{\text{AES}} \rangle .$$

A Gröbner basis computation replaces F_{AES} by another generating set G of the same ideal. Since the ideal is preserved, the solution set is preserved as well:

$$V(F_{\text{AES}}) = V(G).$$

The intended outcome is that the resulting equations reveal the secret-key variables more directly than the original cipher equations.

Small-Scale AES

This chapter introduces the small-scale AES variants used to generate the polynomial systems for the benchmarks in this work. We do not give a full specification of AES, and we do not derive the polynomial equations step by step. The purpose of the chapter is to describe the parameterized cipher family, the round operations that are relevant for the algebraic representation, and the way known plaintext-ciphertext pairs give polynomial systems over $\text{GF}(2)$.

The evaluated Gröbner basis implementations are not specific to AES. They take polynomial systems as input and compute Gröbner bases using their own algorithms and internal representations. Small-scale AES is used in this work because it gives structured and reproducible polynomial systems with adjustable difficulty. This makes it a useful benchmark source, but the conclusions about solver behavior are not limited to AES-derived inputs.

3.1 From AES to Small-Scale AES

The Advanced Encryption Standard (AES) is a standardized symmetric block cipher. It maps a plaintext block and a secret key to a ciphertext block, and decryption uses the same key to recover the plaintext. AES is specified by NIST as a block cipher for electronic data protection. [1]

Standard AES is too large for the purpose of this work. If we used only standard AES instances, the evaluated implementations would likely reach the timeout or memory limit, and the comparison would measure failure behavior. Instead, we use small-scale AES variants. These variants preserve the AES round structure, but allow us to reduce the number of rounds, the state dimensions, and the word size. [4]

Small-scale AES is a parameterized family of AES-like ciphers. It generalizes the AES structure in the sense that AES-128 corresponds to $\text{SR}^*(10, 4, 4, 8)$. The SR^* variant omits MixColumns in the final round, as AES does, while SR keeps MixColumns in the final round as well. The

ciphertexts produced by the two versions are related by an affine mapping, so a solution of the system for one version gives a solution for the other. In this work, we write primarily about $\text{SR}(n, r, c, e)$, because this is the variant used for the benchmark instances. [4]

The original small-scale AES paper defines the family as a framework for analyzing AES-like equation systems and gives an example of deriving such systems using algebraic techniques [4]. We do not repeat that derivation here. A detailed derivation of the systems used in our work is given in [8], and overdefined systems derived from multiple plaintext-ciphertext pairs are studied in [3].

3.2 Parameters of $\text{SR}(n,r,c,e)$

A small-scale AES variant is denoted by $\text{SR}(n, r, c, e)$. The parameter n is the number of encryption rounds, r is the number of rows in the state, c is the number of columns in the state, and e is the word size in bits. The state is an $r \times c$ array of words, and each word contains e bits. The block size is therefore rce bits. The user-provided key has the same size and forms the initial subkey. We present the possible values in Table 3.1. [4]

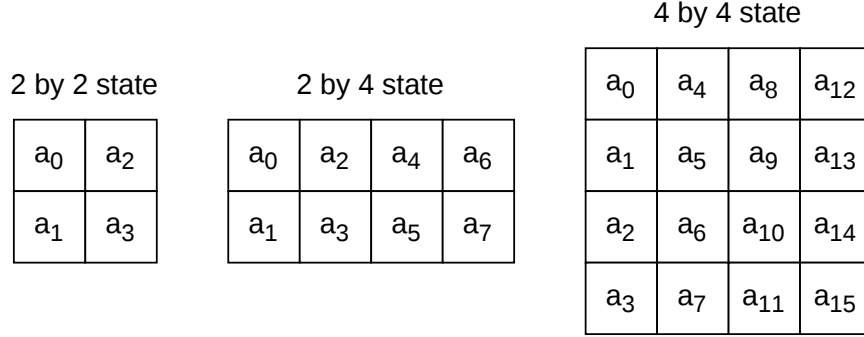
Parameter	Meaning
n	number of encryption rounds, $1 \leq n \leq 10$
r	number of rows in the state, $r \in \{1, 2, 4\}$
c	number of columns in the state, $c \in \{1, 2, 4\}$
e	word size in bits, $e \in \{4, 8\}$

■ **Table 3.1** Parameters of $\text{SR}(n,r,c,e)$

For example, $\text{SR}(2, 2, 2, 4)$ has two rounds, a 2×2 state, and four-bit words. Its block size and key size are $2 \cdot 2 \cdot 4 = 16$ bits. Increasing any of the parameters usually makes the corresponding polynomial systems larger or harder, although the practical difficulty also depends on the number of plaintext-ciphertext pairs and on the concrete data.

The state is numbered column by column, following the AES convention. For example, the word positions in the 2×2 , 2×4 , and 4×4 states are shown in Figure 3.1. This numbering matters because `ShiftRows` acts on rows, while `MixColumns` acts on columns.

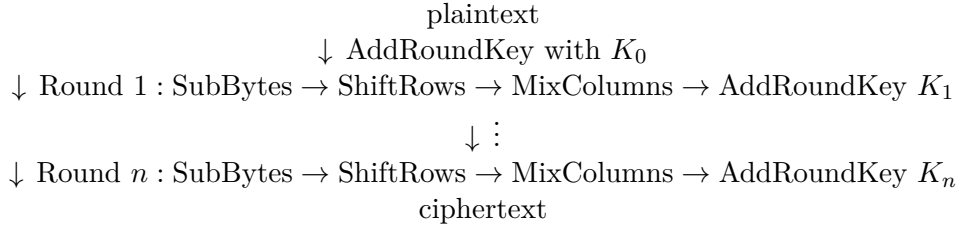
The word size determines the finite field used internally by the cipher. For $e = 4$, words are interpreted as elements of $\text{GF}(2^4)$, defined using the polynomial $X^4 + X + 1$. For $e = 8$, words are interpreted as elements of $\text{GF}(2^8)$, defined using the Rijndael polynomial $X^8 + X^4 + X^3 + X + 1$. This keeps the word operations close to AES while still allowing smaller instances. [4]



■ **Figure 3.1** State numbering used by small-scale AES

3.3 Encryption Process

The encryption structure, shown in Figure 3.2, follows AES. The cipher starts with an initial AddRoundKey operation. Then each round applies SubBytes, ShiftRows, MixColumns, and AddRoundKey. For SR*, MixColumns is omitted in the final round; for SR, it remains present in all rounds. [4]

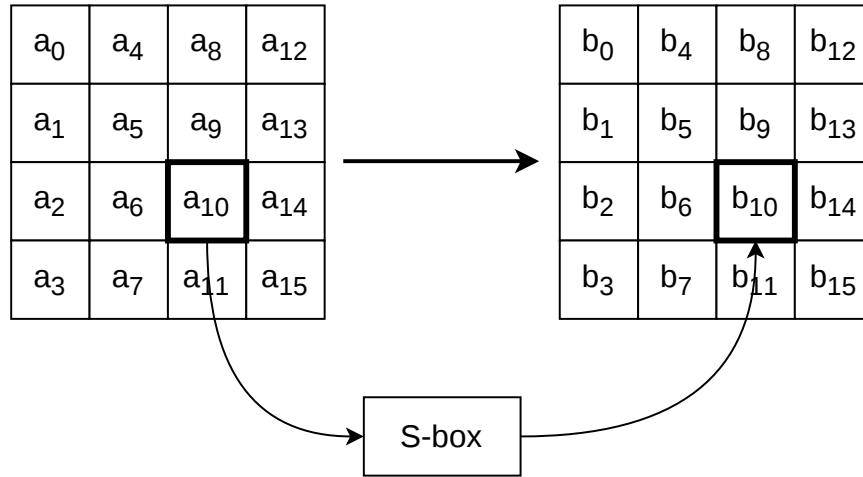


■ **Figure 3.2** High-level encryption structure of SR(n,r,c,e)

The four round transformations have different algebraic roles. SubBytes is nonlinear. ShiftRows is a permutation of state words. MixColumns is a linear transformation over $\text{GF}(2^e)$. AddRoundKey is addition of the subkey, which is XOR at the bit level. All illustrations in the following sections assume $r = c = 4$. The other configurations are analogous and are discussed in [4].

3.3.1 SubBytes

SubBytes applies the S-box independently to every word of the state. For $e = 8$, the S-box is the AES S-box. For $e = 4$, the S-box is defined analogously: inversion in $\text{GF}(2^4)$, with zero mapped to zero, followed by a $\text{GF}(2)$ -linear map and addition of a constant. The SubBytes operation is illustrated in Figure 3.3. [4]



■ **Figure 3.3** AES SubBytes operation, adapted from [9]

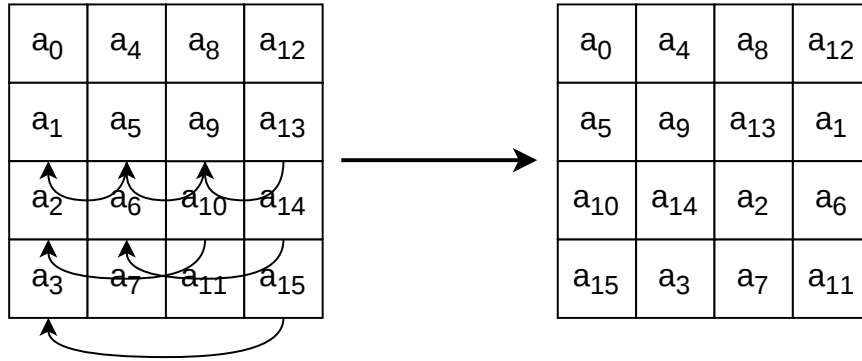
SubBytes is the only nonlinear transformation in an encryption round. The same type of nonlinearity also appears in the key schedule through the S-box used in the key-schedule function. If these S-box applications were replaced by linear transformations, then the encryption and key schedule would consist only of linear or affine operations, and recovering the key from known plaintext-ciphertext pairs would reduce to linear algebra. The S-box is therefore the part that makes the generated polynomial systems nontrivial.

3.3.2 ShiftRows

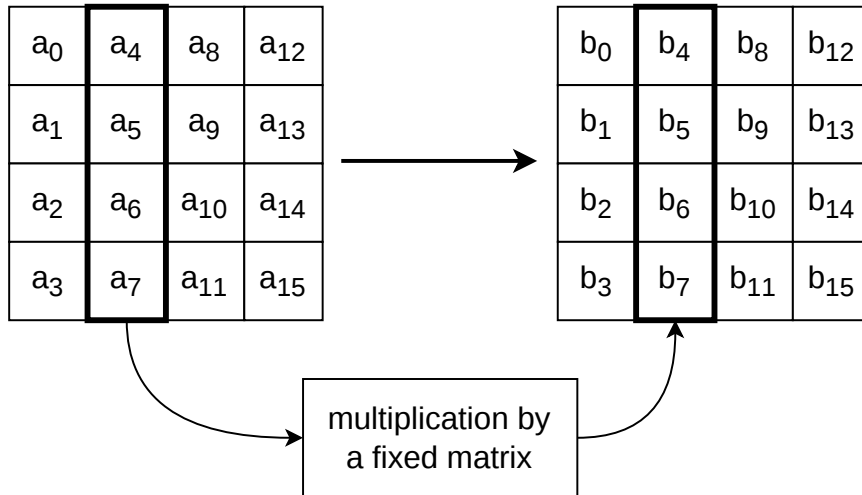
ShiftRows rotates row i of the state to the left by i positions, where rows are indexed from zero. The top row is therefore unchanged. The operation does not change word values; it only changes their positions in the state. If the state has only one row or only one column, the operation has no practical effect. The ShiftRows operation is illustrated in Figure 3.4. [4]

3.3.3 MixColumns

MixColumns acts independently on each column of the state. Each column is multiplied by an invertible circulant matrix over $\text{GF}(2^e)$, with the concrete matrix depending on the number of rows r . For $r = 1$, the matrix is the identity, so the operation has no effect. For $r = 2$ and $r = 4$, it gives the diffusion layer analogous to AES. The MixColumns operation is illustrated in Figure 3.5. [4]



■ **Figure 3.4** AES ShiftRows operation, adapted from [9]

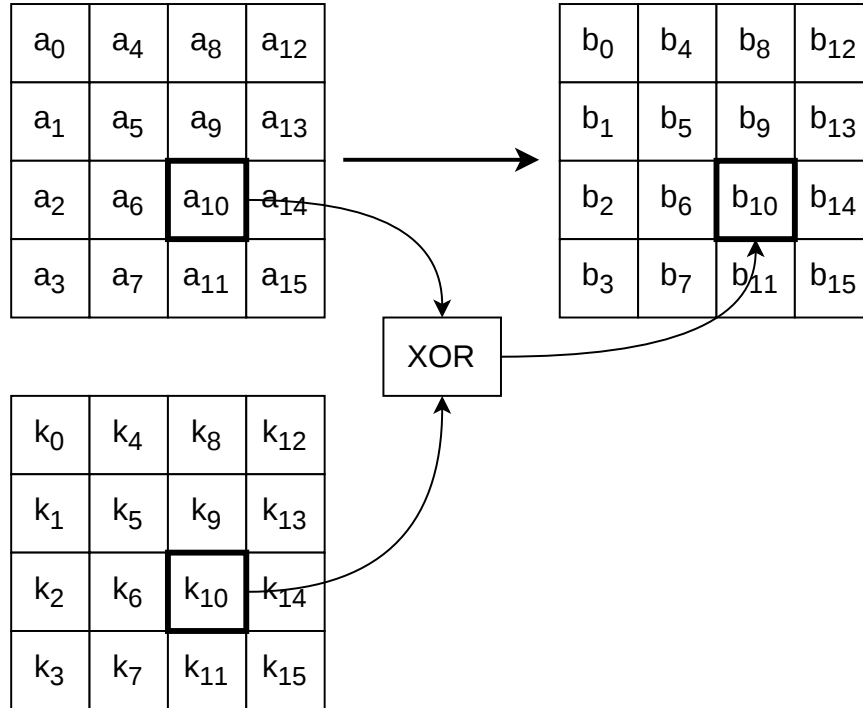


■ **Figure 3.5** AES MixColumns operation, adapted from [9]

3.3.4 AddRoundKey and Key Schedule

For an n -round variant, the key schedule produces $n+1$ subkey blocks. The initial user-provided key has rce bits and forms the initial subkey. AddRoundKey adds the corresponding subkey word to each state word. Since the addition is over a field of characteristic two, this is XOR at the bit level. Small-scale AES begins with an initial AddRoundKey, just as AES does. The AddRoundKey operation is illustrated in Figure 3.6. [4]

The key schedule derives the later subkeys from the initial key. Each subkey has the same $r \times c$ layout as the state and can be viewed as c columns, each containing r words. The key schedule operates column by column. For $c = 4$, the structure follows the AES key schedule and is illustrated in Figure 3.7; for



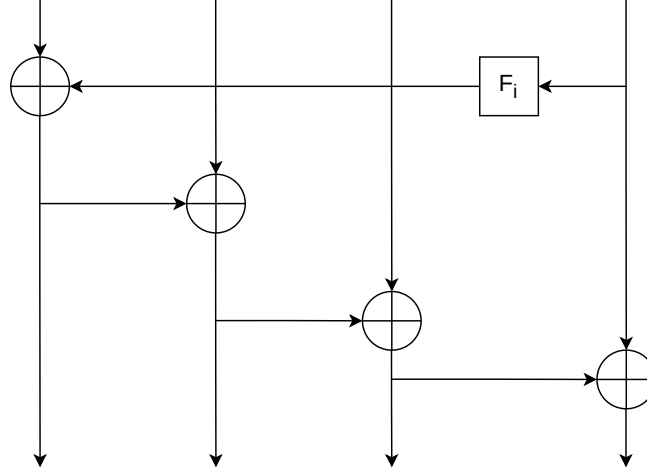
■ **Figure 3.6** AES AddRoundKey operation, adapted from [9]

$c = 1$ and $c = 2$, the structure is reduced accordingly. The nonlinearity comes from the S-box applications inside the function F_i , which also contains a word rotation and a round-dependent constant. [4]

In the polynomial systems used for the benchmarks, the unknowns are the bits of the initial secret key. The subkeys are determined by these bits through the key schedule. Depending on the concrete representation, intermediate state variables may be eliminated so that the final system is expressed only in key variables. This is the approach used by the generator from [8], which we use in this work.

3.4 Known Plaintexts and Polynomial Systems

For a fixed key, a block cipher is a finite map from plaintext blocks to ciphertext blocks. The Universal Mapping Theorem states that any map between finite sets can be represented by polynomial equations over $\text{GF}(p)$, where p is prime. This gives a general reason why ciphers can be described algebraically over finite fields. In this work, we work over $\text{GF}(2)$. [10, p. 190]



■ **Figure 3.7** AES key schedule, adapted from [4]

We work in the known-plaintext setting. This means that we know one or more plaintext-ciphertext pairs encrypted under the same unknown key. For one pair (PT, CT) , the generator constructs a polynomial system over $\text{GF}(2)$ whose solutions are candidate values of the key bits. We can write this as

$$F_{PT,CT}(x_1, \dots, x_m) = 0,$$

where $x_1, \dots, x_m$ are the unknown key bits and $m = rce$ for the selected variant. [11]

The equations obtained from the key schedule depend only on the key variables, while the equations obtained from the encryption also depend on the concrete plaintext and ciphertext values. Consequently, for each plaintext-ciphertext pair we can generate a different polynomial system, even when the same secret key is used. This observation is important for overdefined systems: several systems from different pairs can be combined because they share the same key solution. [8, 3]

If we have plaintext-ciphertext pairs $(PT_1, CT_1), \dots, (PT_t, CT_t)$ encrypted under the same key, then we combine the corresponding systems into one larger system,

$$F_{PT_1,CT_1} \cup F_{PT_2,CT_2} \cup \dots \cup F_{PT_t,CT_t}.$$

This is the source of the overdefined systems used in the experiments. As we will see in Chapter 6, changing the number of plaintext-ciphertext pairs can significantly affect the performance of the evaluated Gröbner basis implementations.

3.5 Use in This Work

The input generator used in this work is based on the scripts from [12] and follows the construction described in [8]. The generated systems use the bits of the initial key as unknowns and are constructed from selected small-scale AES variants and selected plaintext-ciphertext pairs.

Deriving the polynomial system is not the main contribution of this work, and the evaluated solvers do not use any AES-specific information once the input has been generated. In this work, small-scale AES therefore serves primarily as a source of polynomial systems over $\text{GF}(2)$ for experimental evaluation.

Evaluated Implementations

This chapter describes the software systems and concrete Gröbner basis computation implementations evaluated in the following chapters. We define an *implementation* as the full experimental configuration, which consists of:

1. Software system or library
2. Command or algorithmic strategy (including monomial ordering)

There exist many implementations capable of Gröbner basis computation [13, 14]. However, many are not performance-oriented, lack usable documentation, are no longer readily available, or are otherwise unsuitable for benchmarking. We examined the landscape and selected those that seemed most relevant and most likely to produce strong results. To make the comparison more complete, we also evaluated several additional implementations that were already installed in the experimental environment and were sufficiently easy to integrate into the benchmark. In total, we tested 21 such implementations: 20 open-source ones and the reference system Magma [5]. These implementations cover several categories, such as general-purpose computer algebra systems, C/C++ libraries, and language packages.

The common target field is $\text{GF}(2)$. Choosing this field sometimes permits more optimized variants of the algorithms, because the coefficients can be encoded using one bit. Most systems work over the polynomial ring over $\text{GF}(2)$, i.e.,

$$\text{GF}(2)[x_1, \dots, x_n].$$

The remaining systems work over Boolean polynomial rings, i.e.,

$$\text{GF}(2)[x_1, \dots, x_n] / \langle x_1^2 - x_1, \dots, x_n^2 - x_n \rangle.$$

Since we search for solutions to polynomial systems in which each variable corresponds to one bit of the secret key, algorithms that work over Boolean polynomial rings are applicable.

The selected monomial ordering can affect the performance of the algorithms. Where possible, we describe the available monomial orderings for the algorithm as well as the rationale for our choice.

The following sections describe the implementations in detail, grouped by their respective software systems (or libraries). We demonstrate the formats of the inputs and outputs on the following polynomial system:

$$\begin{aligned}x + y + 1 &= 0, \\y + z &= 0, \\xz + z &= 0.\end{aligned}$$

This system has the solution $x = 1$, $y = 0$, and $z = 0$. In the examples, we name the variable y as yy to verify that the system can process variable names longer than one character.

Many of the implementations listed here have an interface in SageMath [15, 16, 17]. However, we discovered that for larger inputs, the interface itself can dominate the runtime of the experiments. For that reason, we work with the implementations directly and do not use this interface. On the other hand, if one does not need to work with large inputs, this interface is easy to use and we recommend it.

4.1 Singular

Singular is a computer algebra system for polynomial computations, with emphasis on commutative and non-commutative algebra, algebraic geometry, and singularity theory. It is free and open-source under the GNU General Public License [18]. We used it in version 4.4.1. [19]

The system exists in two variants – Singular and a C++ library version of Singular called *libSingular* [20]. However, as the kernel of both variants is the same, we use Singular for benchmarking, because it is easier to set up and use, and it is the more accessible variant of Singular.

Upon launching Singular, we are presented with an interactive terminal interface. Here, the user can input expressions using the Singular language [21]. If the interactive mode is not desirable, we can store the same commands in a file and then make Singular execute them non-interactively, printing outputs to the terminal. This is achieved by calling Singular with the file as its first argument. We use this approach for the benchmark runs. Additionally, we use the following terminal options: [22]

`--no-rc` Disables loading the user configuration file.

`-q` Suppresses the startup banner and messages when loading libraries.

`-t` Does not redefine the characteristics of the terminal.

`--cpus=<n>` Sets the number of CPUs used.

`--threads=<n>` Sets the number of worker threads (this option is not listed in the documentation, but is shown when running `Singular --help`).

The documentation lists several algorithms in [23, 24]. We focus on *std*, *slimgb*, *sba*, *groebner*, and *stdhilb*. All of these algorithms work over the polynomial ring.

Singular supports many monomial orderings, such as Lex (lp), Reverse lex (rp), and Degree reverse lex (dp). The documentation recommends using the dp ordering in general. The documentation for the algorithms either also suggests dp or does not specify a preferred ordering. [25, 26]

The input script for Singular is in Code listing 4.1. The first line specifies the ring – GF(2), variables *x*, *yy*, and *z*, using the Degree reverse lex ordering (dp). The following lines define the ideal. Line 6 executes the Gröbner basis calculation. The `<algorithm>` is a placeholder and is to be replaced by one of the five algorithms. This is the only place where the script differs between the algorithms. The final two lines print the result and exit. All of the algorithms produce the same output, which we show in Code listing 4.2.

■ **Code listing 4.1** Singular input example

```

1 ring r = 2,(x,yy,z),dp;
2 ideal I =
3   x+yy+1,
4   yy+z,
5   x*z+z;
6 ideal G = <algorithm>;
7 print(G);
8 quit;
```

■ **Code listing 4.2** Singular output example

```

1 yy+z,
2 x+z+1,
3 z^2
```

Next we give an overview of the different assessed Singular algorithms. All of them return a Gröbner basis, but they differ in their inner workings.

std is a straightforward implementation of the classical Buchberger’s algorithm. To use this algorithm, we replace `<algorithm>` with `std(I)` in Code listing 4.1. [27]

slimgb is designed to keep polynomials slim. The documentation redirects readers to another page with further details. Unfortunately, at the time

of writing, the link does not work. To use this algorithm, we replace `<algorithm>` with `slimgb(I)` in Code listing 4.1. [28]

sba is an optimized version of known signature-based algorithms (such as F5). The function supports additional optional arguments, but evaluating their effect is outside the scope of this work. Therefore, we use the default configuration and replace `<algorithm>` with `sba(I)` in Code listing 4.1. [26]

groebner is a higher-level function that delegates the work to one of the other algorithms. If we do not specify which algorithm to use, the method is chosen heuristically. This does not introduce a new algorithm. Nevertheless, we include it in the experiment, as it is the main function users are expected to use [27]. By including it in the test, we can assess the quality of the heuristic. To use this algorithm, we replace `<algorithm>` with `groebner(I)` in Code listing 4.1. [29]

stdhilb uses the Hilbert-driven Gröbner basis algorithm. Further details are available in [24]. To use this algorithm, we replace `<algorithm>` with `stdhilb(I)` in Code listing 4.1. [24]

4.2 SageMath

SageMath (or Sage for short) is a mathematics software system that builds on top of other existing open-source packages (such as NumPy [30], SciPy [31], and matplotlib [32]). It is free and open-source under the GNU General Public License [18]. We used it in version 10.4. [15]

Sage can be run in two ways. The first is its interactive shell, accessed by launching the `sage` binary from the terminal. It is a customized version of the IPython shell. The second way is to run Python and then import the Sage functions from there, as one would with any other Python package. The documentation provides examples for both variants. We used the latter for our benchmarks. [33]

As mentioned previously, Sage contains the `groebner_basis` function that calculates the Gröbner basis of an ideal. If we exclude all algorithms that come from other binaries, we are left with just three that are part of Sage itself, all of which work over the polynomial ring: `toy:buchberger`, `toy:buchberger2`, and `toy:d_basis`. All of these implementations are marked as toy/educational, meaning they are not optimized and high performance is not expected. [16, 17]

The monomial ordering is chosen during the construction of the polynomial ring. Sage currently supports Lex ordering (lex), Degree lex ordering (deglex), and Degree reverse lex ordering (degrevlex). We use degree reverse lexicographic order in our experiments because it is the default ordering, is generally considered efficient [7], and no alternative ordering is recommended in the documentation. [34]

We present the Python script for Gröbner basis computation in Code listing 4.3. The first line imports the Sage functions, classes, etc. Line 2 constructs the polynomial ring over $\text{GF}(2)$ with the variables named x , yy , and z . As we did not specify the ordering, the ring defaults to Degree reverse lex ordering. Line 3 prepares the variables, and lines 4 to 8 define the ideal. Line 9 computes the Gröbner basis. The `<algorithm>` is the placeholder to be replaced with one of the algorithms. The last two lines print the basis. *toy:buchberger* gives the output in Code listing 4.4, while *toy:buchberger2* and *toy:d_basis* produce the output in Code listing 4.5.

■ **Code listing 4.3** Sage script example

```

1 from sage.all import *
2 R = PolynomialRing(GF(2), ["x", "yy", "z"])
3 x, yy, z = R.gens()
4 I = R.ideal([
5     x + yy + 1,
6     yy + z,
7     x*z + z,
8 ])
9 gb = I.groebner_basis(algorithm='<algorithm>')
10 for g in gb:
11     print(g)

```

■ **Code listing 4.4** Sage output example of *toy:buchberger*

```

1 x*z + z
2 z^2
3 x + yy + 1
4 yy + z

```

■ **Code listing 4.5** Sage output example of *toy:buchberger2* and *toy:d_basis*

```

1 z^2
2 x + z + 1
3 yy + z

```

The three algorithms to replace `<algorithm>` in Code listing 4.3 are as follows:

`toy:buchberger` calls `toy_buchberger.buchberger`, which is an implementation of the original version of Buchberger’s algorithm without Buchberger’s criteria, and the resulting basis is not reduced. [16, 35, 36]

`toy:buchberger2` calls `toy_buchberger.buchberger_improved`. Unlike the first version, this uses the Gebauer–Möller installation to apply Buchberger’s first and second criteria and avoid useless pairs. The resulting basis is reduced. [16, 35, 36]

`toy:d_basis` calls `toy_d_basis.d_basis`. The Sage documentation describes this function as an implementation of the d -basis method from [37]. [16, 38, 36]

4.3 BRiAl

PolyBoRi is a C++ library providing support for Boolean polynomials, exponent vectors, and polynomial rings. Binary decision diagrams are used as the internal storage type for polynomial structures. The documentation also states that this library is now part of Sage. [39]

There exists another project called BRiAl. Its GitHub repository¹ describes it as a “Successor to PolyBoRi”, whereas its README states that “BRiAl is the legacy version of PolyBoRi maintained by sagemath developers”. These descriptions suggest different relationships between BRiAl and PolyBoRi: the former presents BRiAl as a successor, while the latter presents it as a legacy version. However, since:

1. the Sage documentation page for BRiAl states that it is the successor to PolyBoRi, [40]
2. the last update of PolyBoRi is from 2013, [41]
3. the last update of BRiAl is from 2025, [42]
4. and we could not find any further authoritative clarification about the PolyBoRi/BRiAl relationship,

we concluded that BRiAl is the relevant actively maintained codebase and therefore chose it for this work. It is free and open-source under the GNU General Public License [18]. We used version 1.2.15. [42]

Again, we focused directly on the C++ library, skipping the Sage interface. To the best of our knowledge, no Gröbner-basis-related documentation for the C++ interface exists. Nevertheless, we managed to construct a working program using this library.

¹<https://github.com/BRiAl/BRiAl>

As the project description says, this is one of the few implementations that work over the Boolean polynomial ring. The source code contains five monomial orderings: `LexOrder`, `DegLexOrder`, `DegRevLexAscOrder`, `BlockDegLexOrder`, and `BlockDegRevLexAscOrder`. The constructor of `BoolePolyRing`, which we use, takes the monomial ordering as a parameter, with the default value `LexOrder`. As we did not find any evidence suggesting one of the other orderings, we use the default `LexOrder`. [43, 44]

We provide an example of C++ source code for Gröbner basis computation in Code listing 4.6. The first 8 lines import the required libraries and create aliases for better readability. Line 10 constructs a Boolean polynomial ring with 3 variables. Lines 11 to 16 register the variables and create local references to them. Lines 17 to 23 add the polynomials. Lines 24 and 25 calculate the basis, and the remaining lines print it. The output of this program is shown in Code listing 4.7.

4.4 CoCoALib

CoCoA System is a program for computations with numbers and polynomials. Specifically, it offers functions for computations in commutative algebra and related fields. It is free and open-source under the GNU General Public License [18]. We used version 0.99880. [45, 46]

The CoCoA System consists of two parts – CoCoA-5 and CoCoALib. CoCoA-5 is an interactive system that runs in the terminal. It uses its own programming language, which is documented in [47]. CoCoALib is the kernel of CoCoA-5, written in C++. As with many of the other implementations, we focus on the library version. [46]

This library works over the polynomial ring. According to the documentation, it supports four monomial orderings: Degree reverse lex (`DegRevLex`), Degree lex (`DegLex`), Lex (`Lex`), and the so-called Xel ordering (`Xel`). The default ordering when creating a ring is Degree reverse lex. The authors of [48] state that this ordering is known to be the “best general choice”. We therefore chose to use it as well. [49]

We present an example usage of the library in Code listing 4.8. The first four lines import the required libraries and use the `CoCoA` namespace to make the remaining code cleaner. Line 5 creates a global manager. This object manages global values and settings used by CoCoALib, and it must be created before using other CoCoALib features. Lines 6 to 8 create the polynomial ring over $\text{GF}(2)$ with variables named x , yy , and z . As we did not specify any monomial ordering, the default Degree reverse lex ordering is used. Lines 9 to 11 create local references to the variables, and lines 12 to 17 create the ideal. Finally, line 18 computes and prints the basis. The output of this program is shown in Code listing 4.9. [50, 51, 52]

■ **Code listing 4.6** BRiAl source code example

```

1  #include <polybori.h>
2  #include <polybori/groebner/GroebnerStrategy.h>
3  #include <iostream>
4  #include <vector>
5  using Ring = polybori::BoolePolyRing;
6  using Poly = polybori::BoolePolynomial;
7  using Var = polybori::BooleVariable;
8  namespace pbg = polybori::groebner;
9  int main() {
10     Ring ring(3);
11     ring.setVariableName(0, "x");
12     ring.setVariableName(1, "yy");
13     ring.setVariableName(2, "z");
14     Var x(0, ring);
15     Var y(1, ring);
16     Var z(2, ring);
17     Poly f1 = Poly(x) + Poly(y) + Poly(true, ring);
18     Poly f2 = Poly(y) + Poly(z);
19     Poly f3 = Poly(x * z) + Poly(z);
20     pbg::GroebnerStrategy strat(ring);
21     strat.addAsYouWish(f1);
22     strat.addAsYouWish(f2);
23     strat.addAsYouWish(f3);
24     strat.symmGB_F2();
25     std::vector<Poly> gb = strat.minimalizeAndTailReduce();
26     for (const Poly& g : gb) {
27         std::cout << g << '\n';
28     }
29     return 0;
30 }

```

■ **Code listing 4.7** BRiAl output example

```

1  x + 1
2  yy
3  z

```

4.5 Giac

Giac is a C++ library for computer algebra, supporting arithmetic, integration, differentiation, limits, and others. It is free and open-source under the GNU General Public License [18]. We tested it in version 1.9.0. [53]

■ **Code listing 4.8** CoCoALib source code example

```

1  #include <iostream>
2  #include <CoCoA/library.H>
3  int main() {
4      using namespace CoCoA;
5      GlobalManager CoCoAFoundations;
6      SparsePolyRing P = NewPolyRing(
7          NewZZmod(2), symbols("x,yy,z")
8      );
9      RingElem x = indet(P, 0);
10     RingElem y = indet(P, 1);
11     RingElem z = indet(P, 2);
12     std::vector<RingElem> gens = {
13         x + y + 1,
14         y + z,
15         x*z + z,
16     };
17     ideal I(P, gens);
18     std::cout << GBasis(I) << '\n';
19     return 0;
20 }

```

■ **Code listing 4.9** CoCoALib output example

```

1  [yy +z,  x +z +1,  z^2]

```

The default use scenario of Giac is via Xcas, which is a graphical interface for Giac. The Xcas installation contains a text version called **giac**, which can be launched from the terminal. This gives us an interactive terminal interface. Examples of the syntax and possible calculations are documented in [54]. Launching the **giac** binary with a filename as an argument makes it execute the commands from the file instead. As for other command line parameters, we were not able to find any mention of them in the documentation, and neither **giac -h** nor **giac --help** works. Therefore, we use this binary with the filename as its first argument and with no other parameters. [53, 55]

The input we present works over the polynomial ring. We did not find any mention of Boolean polynomial ring support in Giac. To calculate the Gröbner basis, we use the **gbasis** command. This command takes two mandatory and three optional arguments. The two mandatory arguments are a list of polynomials and a list of variables. Two of the optional arguments are **with_cocoa**, which enables Giac to use CoCoALib to calculate the basis, and **with_f5**,

which says whether to use the F5 algorithm of CoCoALib. As we are testing CoCoALib separately, we decided to set both options to **false**. [56]

The last optional argument selects the monomial ordering to be used. It currently supports three orderings: Lex (plex), Degree lex (tdeg), and Reverse lex (revlex). The documentation states that plex is the default. However, the authors of [57] used revlex instead. We therefore decided to test this implementation twice, with both plex and revlex.

The input for Giac that we use is shown in Code listing 4.10. The first line sets the number of threads. Lines 2 to 4 construct the polynomials, and line 5 puts them in a list and makes them modulo 2, effectively making them polynomials over GF(2). Line 6 enumerates the variable names. Line 7 calculates and prints the Gröbner basis. The `<ordering>` on this line is to be replaced with the ordering, i.e., **plex** or **revlex**. Giac prints the result of each line. To make parsing easier (as described in Chapter 5), we finish all lines with `;;`. This makes the lines print "Done", which is easier to filter out. This is correct on line 7 as well, because we do not care about the return value of the **print** command itself, only about what it prints, which is still present. We present the output of the revlex run in Code listing 4.11. The plex run outputs the basis `[z2,yy-z,x-z-1]` (the remaining lines are identical).

■ **Code listing 4.10** Giac input example

```

1 threads:=48;;
2 f1:=x+yy+1;;
3 f2:=yy+z;;
4 f3:=x*z+z;;
5 F:=[f1,f2,f3]%2;;
6 X:=[x,yy,z];;
7 print(gbasis(F,X,<ordering>,with_cocoa=false,with_f5=false));;
```

4.6 msolve

msolve is a C library implementing computer algebra algorithms for solving polynomial systems using advanced Gröbner basis techniques. It is free and open-source under the GNU General Public License [18]. We tested version 0.9.4. [58, 6, 59]

The msolve library comes with an **msolve** binary, which can be run from the terminal. Unlike many of the other systems, it does not support an interactive mode of operation. Instead, the binary takes a file defining a polynomial system and writes the result to another file. This is ideal for benchmarking. The binary can take multiple arguments. The ones we used are: [58, 60]

-g 2 Requests computation of the full reduced Gröbner basis.

■ **Code listing 4.11** Giac output example with revlex as the monomial ordering

```

1 // Using locale /usr/share/locale/
2 // en_US.UTF-8
3 // /usr/share/locale/
4 // giac
5 // UTF-8
6 // Maximum number of parallel threads 48
7 // Unable to find keyword file /usr/share/giac/doc/en/keywords
8 Added 0 synonyms
9 "Done",
10 // Time 0
11 "Done",
12 // Time 0
13 "Done",
14 // Time 0
15 "Done",
16 // Time 0
17 "Done",
18 // Time 0
19 "Done",
20 // Time 0
21 [z^2,x+z+1,yy+z]
22 "Done"
23 // Time 0.01
24 // Total time 0.01

```

-t <n> Sets the number of threads used.

-f <file> Specifies the input file.

-o <file> Specifies the output file.

The library uses the Degree reverse lex monomial ordering, with no option to select another ordering, and it works over the polynomial ring. According to the documentation, msolve builds on the F4 algorithm. [60]

We present the input file format in Code listing 4.12. The first line specifies the variable names. The second line describes the field, i.e., GF(2). The remaining lines consist of the polynomials, all of which except the last are terminated by a comma. The output of this input is shown in Code listing 4.13. [58]

■ **Code listing 4.12** msolve input example

```

1 x,yy,z
2 2
3 x+yy+1,
4 yy+z,
5 x*z+z

```

■ **Code listing 4.13** msolve output example

```

1 #Reduced Groebner basis data
2 #---
3 #field characteristic: 2
4 #variable order:      x, yy, z
5 #monomial order:      graded reverse lexicographical
6 #length of basis:     3 elements
7 #---
8 [1*yy^1+1*z^1,
9 1*x^1+1*z^1+1,
10 1*z^2]:

```

4.7 Macaulay2

Macaulay2 is a software system devoted to supporting research in algebraic geometry and commutative algebra. It supports Gröbner basis computation over polynomial rings. It is free and open-source under the GNU General Public License [18]. We used version 1.25.11. [61, 62]

This system comes in the form of a binary called M2. Upon launching it from the command line, we are presented with an interactive prompt, where each input yields output. Macaulay2 has its own language, which is described in [63]. Of the command line options, we used only `--script <file>`, which executes the commands from the specified file, prints the outputs, and exits. [64, 65]

We tested four algorithms: the default command *gb*, *MGB*, *F4*, and *BooleanGB*. The system works over polynomial rings for the first three implementations and over a Boolean polynomial ring for *BooleanGB*. [66, 67, 68]

Many monomial orderings are supported by Macaulay2, including Lex (Lex), Graded lex (GLex), and Graded reverse lex (GRevLex). The documentation states that GRevLex is the default order, in large part because it is often the most efficient order for use with Gröbner bases. The only exception here is *BooleanGB*, which uses the Lex order. [69, 70, 68]

The input and output files differ slightly depending on whether we calculate over a polynomial ring or a Boolean polynomial ring. We first focus on the polynomial ring case in Code listing 4.14. The first line defines the ring over the field $\mathbb{Z}/2\mathbb{Z}$, which is expressed as `ZZ/2`. We use this notation instead of `GF(2)` because the documentation for the experimental MGB and F4 strategies describes finite fields in Macaulay2 as `ZZ/p`; in our case this is `ZZ/2`. It then defines the three variables and selects `GRevLex` ordering. Lines 2 to 6 define the ideal. Line 7 calculates the Gröbner basis, where `<algorithm>` is a placeholder to be replaced with one of the algorithms described below. The last two lines print the result and exit. The output of this script is shown in Code listing 4.15. [66]

■ **Code listing 4.14** Macaulay2 input example over polynomial ring

```

1 R = ZZ/2[x, yy, z, MonomialOrder => GRevLex];
2 I = ideal(
3     x+yy+1,
4     yy+z,
5     x*z+z
6 );
7 G = <algorithm>;
8 print G;
9 exit 0;
```

■ **Code listing 4.15** Macaulay2 output example over polynomial ring

```

1 | yy+z x+z+1 z2 |
```

The input for the Boolean polynomial ring is shown in Code listing 4.16. It is prepended by a line that imports the required package. Line 2 uses `Lex` ordering instead of `GRevLex`. Line 8 uses a different command to calculate the Gröbner basis, and line 9 first extracts the generators of the resulting ideal before printing them. The output is shown in Code listing 4.17. [68]

Next we outline the differences between the four algorithms:

gb is the default algorithm of Macaulay2 for Gröbner basis computation. With no `Strategy` option, `groebnerBasis` simply calls `gb`. To use it, we replace `<algorithm>` in Code listing 4.14 with `groebnerBasis(I)`. [66, 67]

MGB comes from the `mathicGB` package, which is based on the fast data structures from `mathic`. `Mathic` is a C++ library of fast data structures

■ **Code listing 4.16** Macaulay2 input example over Boolean polynomial ring

```

1 needsPackage "BooleanGB";
2 R = ZZ/2[x, yy, z, MonomialOrder => Lex];
3 I = ideal(
4     x+yy+1,
5     yy+z,
6     x*z+z
7 );
8 G = gbBoolean I;
9 print gens G;
10 exit 0;

```

■ **Code listing 4.17** Macaulay2 output example over Boolean polynomial ring

```

1 | yy x+1 z |

```

designed for use in Gröbner basis computation. Further details are available in [71]. To use it, we replace `<algorithm>` in Code listing 4.14 with `groebnerBasis(I, Strategy => "MGB")`. [66, 72, 73]

F4 is the experimental `Strategy => "F4"` exposed by `groebnerBasis`. The documentation describes it as a Faugère-like algorithm, but it does not document it in enough detail for a more specific classification here. To use it, we replace `<algorithm>` in Code listing 4.14 with `groebnerBasis(I, Strategy => "F4")`. [66]

BooleanGB is the only algorithm of the four that comes from a separate package of the same name. It is also the only one that works over a Boolean polynomial ring and uses Lex ordering. The Boolean polynomial ring allows it to be implemented bitwise, which reduces the computational complexity. [68]

4.8 M4GB

M4GB is an efficient algorithm for computing Gröbner bases. Like F4, it is an extension of Buchberger's algorithm. Compared to F4, it claims to further reduce redundant work in the processing of reducible monomials. The implementation of M4GB is free and open-source under the GNU General Public License [18]. As far as we can tell, the project does not specify an explicit version number. For reproducibility, we used the last commit, i.e., 0a166c87. [74, 75]

This implementation comes in the form of a script called `solver.sh`. The script takes a file specifying the input and writes the Gröbner basis to another file. Upon launching, the script takes the number of variables and the size of the field and builds a custom binary for this combination. It then calls this binary, which performs the computation itself. If the binary already exists, it is reused and not compiled again. If we want to create the binary beforehand ourselves, we can do so by calling `make MAXVARS=3 FIELD_SIZE=2 m4gb` (for three variables and a polynomial ring over $\text{GF}(2)$). [74]

As for the other parameters, the README is outdated. We therefore rely on the options shown by calling `./solver.sh -h`. This shows that the script takes the input file as the first positional argument, and optionally the output file as the second. We also use all of the optional parameters:

- `-s m4gb` Compile and run M4GB (the script is also capable of compiling and running OpenF4 and FGb).
- `-v <n>` Specifies the verbosity level (we use 0).
- `-n <n>` Sets the number of variables.
- `-f <n>` Sets the size of the finite field.
- `-t <n>` Specifies the number of threads.

This implementation works over the polynomial ring and uses the Degree reverse lex monomial ordering, which is also the only one supported. [74, 75]

An example input is presented in Code listing 4.18. The first line selects $\text{GF}(2)$, and the second line specifies the variables. The remaining lines define the polynomials. The output is shown in Code listing 4.19.

■ **Code listing 4.18** M4GB input example

```

1 $fieldsize 2
2 $vars x yy z
3 x+yy+1
4 yy+z
5 x*z+z

```

■ **Code listing 4.19** M4GB output example

```

1 z
2 yy
3 x + 1

```

4.9 Groebner.jl

Julia is a high-level, high-performance dynamic language for technical computing. It is free and open-source under the MIT License [76]. We used version 1.12.5. [77, 78]

Groebner.jl is a Julia package for Gröbner basis computation written in Julia. It implements the F4 algorithm. The authors also acknowledge that they adapt some code from msolve. It is free and open-source under the GNU General Public License [18]. We used version 0.10.3. [79, 80]

Like many dynamic languages, Julia can be used in two ways – interactively via a REPL², or non-interactively by executing a file. Again, the latter is the useful mode for us. Among the available command line parameters, we are interested in the following: [81, 82]

`--startup-file=no` Do not load the startup file.

`--threads <n>` Enables multithreading.

This package works over the polynomial ring. It also supports multiple monomial orderings, mainly Lex (Lex), Degree lex (DegLex), and Degree reverse lex (DegRevLex). We did not find any recommendation in the documentation to prefer one over the others. However, as DegRevLex is used frequently in the documentation examples and is also the best default choice for many of the other implementations, we chose it. [83]

To construct the polynomials, Groebner.jl supports multiple frontends. It also supports a low-level interface that allows us to specify the polynomials as lists of coefficients and exponent vectors. While this is less human-readable than the other variants, it allows us not to depend on another frontend. This reduces the number of required packages and can also reduce the overhead of polynomial construction, although this overhead is likely negligible compared to the subsequent Gröbner basis computation. We demonstrate this low-level interface on the following example, which assumes that x , y , and z are represented by the first, second, and third positions of the exponent vectors, respectively: [84]

$x + y + 1 = 1 \cdot x^1 \cdot y^0 \cdot z^0 + 1 \cdot x^0 \cdot y^1 \cdot z^0 + 1 \cdot x^0 \cdot y^0 \cdot z^0$. The coefficients are $[1, 1, 1]$ and the exponent vectors are $[[1, 0, 0], [0, 1, 0], [0, 0, 0]]$,

$y + z = 1 \cdot x^0 \cdot y^1 \cdot z^0 + 1 \cdot x^0 \cdot y^0 \cdot z^1$. The coefficients are $[1, 1]$ and the exponent vectors are $[[0, 1, 0], [0, 0, 1]]$,

$xz + z = 1 \cdot x^1 \cdot y^0 \cdot z^1 + 1 \cdot x^0 \cdot y^0 \cdot z^1$. The coefficients are $[1, 1]$ and the exponent vectors are $[[1, 0, 1], [0, 0, 1]]$,

²read-eval-print loop, i.e., an interactive shell

We present an example input in Code listing 4.20. The first line loads the package. The second line constructs the polynomial ring over $\text{GF}(2)$, using `DegRevLex` and 3 variables. Lines 3 to 6 define the polynomials using the lists of monomials and coefficients. As we are working over $\text{GF}(2)$, the coefficients are always ones. Lines 7 to 9 compute the Gröbner basis using parallelization. The remaining lines print the result. The output is shown in Code listing 4.21 (the lines have been wrapped; the first six lines are output as one line). This output corresponds to the polynomials $y + z$, $x + z + 1$, and z^2 . [83]

■ **Code listing 4.20** Groebner.jl input example

```

1 using Groebner
2 ring = Groebner.PolyRing(3, Groebner.DegRevLex(), 2)
3 monoms = [[[1, 0, 0], [0, 1, 0], [0, 0, 0]],
4           [[0, 1, 0], [0, 0, 1]],
5           [[1, 0, 1], [0, 0, 1]]]
6 coeffs = [[1, 1, 1], [1, 1], [1, 1]]
7 gb_monoms, gb_coeffs = Groebner.groebner(
8     ring, monoms, coeffs; tasks=48
9 )
10 println(gb_monoms)
11 println(gb_coeffs)

```

■ **Code listing 4.21** Groebner.jl output example (line wrapped)

```

1 Vector{Vector{UInt32}}[[[0x00000000, 0x00000001, 0x00000000],
2                        [0x00000000, 0x00000000, 0x00000001]],
3                        [[0x00000001, 0x00000000, 0x00000000],
4                        [0x00000000, 0x00000000, 0x00000001],
5                        [0x00000000, 0x00000000, 0x00000000]],
6                        [[0x00000000, 0x00000000, 0x00000002]]]
7 Vector{UInt8}[[0x01, 0x01], [0x01, 0x01, 0x01], [0x01]]

```

4.10 openf4

The `openf4` library is a C++ implementation of the F4 algorithm. It is free and open-source under the GNU General Public License [18]. We evaluated version 1.0.1. [85]

The library primarily works on prime fields of characteristic lower than 2^{32} and on $\text{GF}(2)$ extensions of degree less than 64. The library can be compiled with the following libraries: [85]

Givaro which allows `openf4` to calculate over prime fields of characteristic higher than 2^{32} . [86]

FFLAS-FFPACK which can make `openf4` slightly faster at the cost of increased memory for some very specific systems. [87]

We did not use Givaro, as we need only $\text{GF}(2)$, and the documentation describes the benefit of FFLAS-FFPACK as limited to some very specific systems, so we did not use it either. We compiled the library with the default configuration options, i.e., with optimizations, vectorization, and OpenMP support for parallelization. [85]

`openf4` works over the polynomial ring. The documentation consulted for this work does not state the monomial ordering explicitly, and the code path we use does not expose it as a user-selectable parameter. [88]

The library contains many C++ source code examples of its usage, but it does not come with a binary that we could execute directly. Therefore, we modified one of the examples to make our own solver. [89]

We present the example source code adapted from [89] in Code listing 4.22. The first four lines include the library headers and namespace directives. Lines 5 to 9 configure `openf4` to disable verbosity and set multithreading. Line 13 sets the number of variables, and lines 14 to 18 configure the variable names. Lines 19 to 23 construct the ideal from polynomials over $\text{GF}(2)$. Lines 24 and 25 call the F4 algorithm and retrieve the reduced Gröbner basis. The remaining lines print the output and exit. The output can be seen in Code listing 4.23.

4.11 Magma

Magma is the only proprietary system discussed in this work. We include it as a reference point representing state-of-the-art commercial software for Gröbner basis computation. It is a large software package designed for computations in algebra, number theory, algebraic geometry, and algebraic combinatorics. We used version V2.28-15. [5]

It is a command line program that drops the user into a REPL, which uses its own language. An overview of the language can be found in [90]. However, we want to execute the commands from a file instead. We can achieve this by calling Magma as `magma -b <file>`. This will execute the commands and print the result. [91]

Magma supports both ordinary polynomial rings and Boolean polynomial rings. We use the Boolean polynomial ring, as it is natural for our use case and is more specialized. Three monomial orderings are available when creating the Boolean polynomial ring: Lex (lex), Graded lex (glex), and Graded reverse lex (grevlex). The documentation recommends grevlex for Gröbner basis computation. Magma uses the F4 algorithm. [92, 93, 94, 95]

■ **Code listing 4.22** openf4 source code example

```

1  #include <iostream>
2  #include <openf4.h>
3  using namespace F4;
4  using namespace std;
5  int F4::VERBOSE = 0;
6  #ifdef USE_OPENMP
7  int F4::NB_THREAD = omp_get_num_procs();
8  #else
9  int F4::NB_THREAD = 1;
10 #endif
11 int main(int argc, char **argv)
12 {
13     Monomial::initMonomial(3);
14     string * vars = new string[3];
15     vars[0] = "x";
16     vars[1] = "yy";
17     vars[2] = "z";
18     Monomial::setVariable(vars);
19     vector<Polynomial<ElementGF2>> polynomialArray;
20     polynomialArray.emplace_back("x+yy+1");
21     polynomialArray.emplace_back("yy+z");
22     polynomialArray.emplace_back("x*z+z");
23     Ideal<ElementGF2> I(polynomialArray, 3);
24     I.f4();
25     vector<string> basis = I.getReducedGroebnerBasis();
26     for (const auto & f : basis)
27     {
28         cout << f << '\n';
29     }
30     return 0;
31 }

```

■ **Code listing 4.23** openf4 output example

```

1  (1*x^1) + (1*z^1) + (1*1)
2  (1*yy^1) + (1*z^1)
3  (1*z^2)

```

We present an example input in Code listing 4.24. The first two lines set parallelization and disable computation on the GPU. Using the GPU would

give Magma an advantage over the other implementations. Line 3 constructs the Boolean polynomial ring with three variables using grevlex ordering. Lines 4 to 8 construct the polynomials. Line 9 computes the Gröbner basis, and the remaining lines print it and exit. The output of this script is shown in Code listing 4.25.

■ **Code listing 4.24** Magma input example

```
1 SetGPU(false);
2 SetNthreads(48);
3 P<x, yy, z> := BooleanPolynomialRing(3, "grevlex");
4 B := [
5     x+yy+1,
6     yy+z,
7     x*z+z
8 ];
9 G := GroebnerBasis(B);
10 print G;
11 quit;
```

■ **Code listing 4.25** Magma output example

```
1 [
2     x + 1,
3     yy,
4     z
5 ]
```

Benchmarking Setup

The main part of this work is to benchmark the implementations. We want to assess not only whether they work and are capable of Gröbner basis computations, but also measure metrics such as the time it takes them to do so, memory consumption, and others. We created our own benchmarking framework to accomplish this task. This chapter is devoted to describing this framework. All tests were conducted on a machine running the *Ubuntu 24.04.6 LTS* operating system, equipped with two *Intel Xeon Gold 6136* processors with 12 cores each. In total, the machine has 24 physical cores, 48 logical cores, and 754 GiB of RAM. This framework utilizes certain mechanisms of the Linux kernel, and is therefore compatible only with Linux-based operating systems.

5.1 Architecture Overview

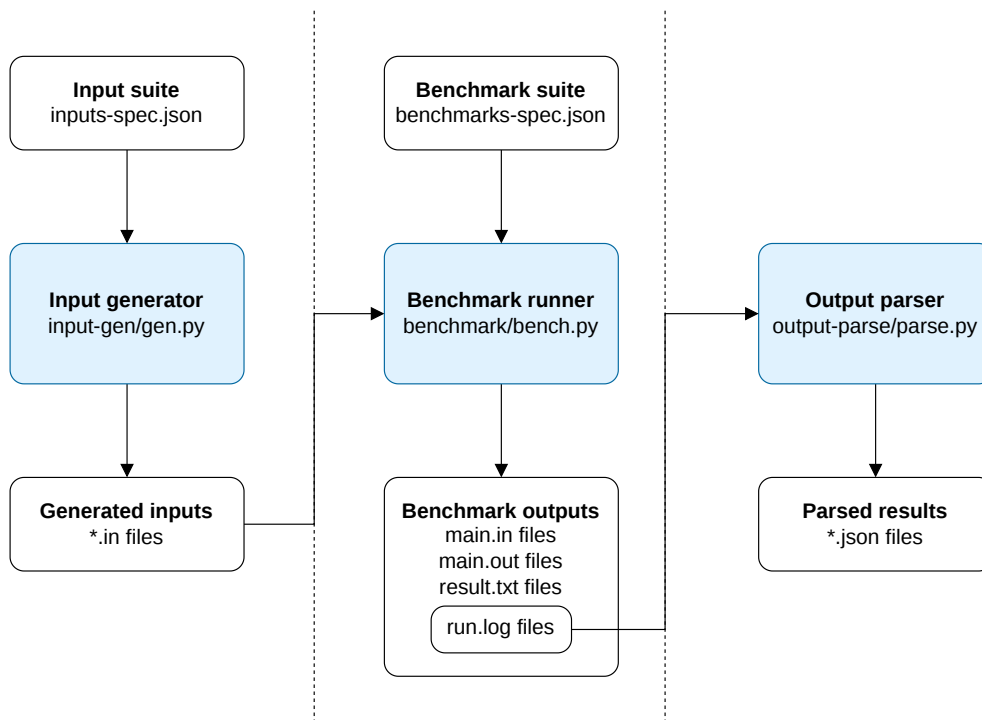
The framework is divided into three phases:

Input generator takes a specification of the input suite. This specifies which small-scale AES variants should be generated, how many plaintext-ciphertext pairs to use, how many keys to generate, and whether to generate the Boolean polynomials. The resulting files contain the polynomials corresponding to the small-scale AES variants for the specified keys, which the evaluated implementations are then run on.

Benchmark runner is the main and most complex phase of the framework. It takes a specification of the benchmark suite. This specifies which implementations should be run and on which inputs from the previous phase. It also allows specifying the parallelization and timeout. It outputs multiple files, including the Gröbner basis found by the implementation for the given input instance and a `run.log` file that contains the measured data, such as execution duration and consumed memory.

Output parser collects the `run.log` files and transforms them into structured JSON files. This format allows the data to be processed further, e.g., for data visualization, table creation, etc.

Each phase of the framework works on the files output by the previous phase. This means that the phases are to be run one after the other and are independent of each other. This allows users, for example, to generate their own input files and skip the Input generator. For instance, we used such custom-constructed input files in Chapter 4, because the Input generator names the variables `k000`, `k001`, etc., but we wanted `x`, `yy` and `z` for better readability for the examples. We provide a diagram of the framework phases in Figure 5.1.



■ **Figure 5.1** High-level overview of the benchmarking framework

The whole framework is written entirely in Python [96]. We made this choice because Python offers a large number of libraries for tasks such as JSON parsing, subprocess invocation, and logging, all of which we use. One downside of Python is that it is slower compared to some other languages. However, this does not limit the framework, because it mainly measures the performance of other software and does not perform intensive computations in Python itself.

We also built the Benchmark runner around Python’s Asynchronous I/O (also known as `asyncio`) [97]. This enables us to write concurrent code without

having to deal with threads and their synchronization. Although `asyncio` is not suitable for CPU-bound tasks, the heavy Gröbner basis computation takes place in the benchmarked software, while the Benchmark runner “only” orchestrates external processes. This orchestration is therefore I/O-bound, which is the use case `asyncio` is designed for.

The following sections focus on these phases in more detail.

5.2 Input Generator

We first establish terminology related to input files. We define four terms:

Input variant is a concrete small-scale AES variant with a defined number of PT-CT pairs, e.g., $\text{SR}(2, 2, 2, 4)$ for 4 PT-CT pairs.

Input instance is a list of variables and polynomials generated for a selected input variant and for one key, stored in a `.in` file. These are the files generated by this phase.

Input is a named set of input variants together with information on how many keys the input instances are generated for and whether the Boolean polynomials should be included.

Input suite is a list of inputs. These are stored in the `inputs-spec.json` file.

An example of an input suite specified in a `inputs-spec.json` is available in Code listing 5.1. This defines two inputs, named *pilot* and *main*. The *pilot* input has 8 keys and does not include the Boolean polynomials $k_i^2 + k_i$. It also defines four SR variants and two PT-CT pair counts. The Input generator makes a Cartesian product of the SR variants and the pair counts. In this case, this results in $2 \cdot 4 = 8$ input variants. The Input generator then generates 8 keys for each of these variants, constructs the polynomial systems and stores them in `.in` files, resulting in $8 \cdot 8 = 64$ input instances for the *pilot* input. Similarly, the *main* input describes $2 \cdot 1 = 2$ input variants and $2 \cdot 10 = 20$ input instances, all of which include the Boolean polynomials.

This phase of the framework is launched by calling `gen.py`. It reads the input suite from `inputs-spec.json`, and constructs the input instances. This polynomial system generation is based on the work of Marek Bielik and coauthors [8, 12]. The methods they provide take the small-scale AES variant and the number of PT-CT pairs and generate the polynomials. These input instances are then stored in the form of `.in` files.

`gen.py` generates the input instances for all inputs from the input suite. Alternatively, we can pass one or more input names as positional arguments to generate input instances only for them. The script does not regenerate input

■ Code listing 5.1 `inputs-spec.json` example

```
1  [  
2    {  
3      "name": "pilot",  
4      "keys": 8,  
5      "boolean": false,  
6      "variants": [  
7        {  
8          "pairs_per_key": [2, 8],  
9          "SRs": [  
10             [1, 2, 4, 8],  
11             [1, 4, 2, 8],  
12             [1, 4, 4, 8],  
13             [2, 2, 2, 4]  
14         ]  
15     }  
16 ]  
17 },  
18 {  
19   "name": "main",  
20   "keys": 10,  
21   "boolean": true,  
22   "variants": [  
23     {  
24       "pairs_per_key": [16],  
25       "SRs": [  
26         [1, 2, 4, 4],  
27         [2, 2, 4, 4]  
28     ]  
29   }  
30 ]  
31 }  
32 ]
```

instances that already exist. This means that if the script is terminated prematurely, it continues from where it left off on the next invocation. Additionally, it takes the following parameters:

--inputs-spec <file> is the path to the `inputs-spec.json` file. It defaults to `./inputs-spec.json`.

--inputs-folder <directory> is the path to the directory where the resulting `.in` files are stored. It defaults to `./inputs`.

The generated input instances are stored in `<inputs-folder>/<input>/SR_<n>_<r>_<c>_<e>:pairs_<pair-count>/<key-index>.in`. For example, the first input instance from the *pilot* input would be stored in `./inputs/pilot/SR_1_2_4_8:pairs_2/0.in`.

Finally, we present an example of the input instance file in Code listing 5.2. The lines before `=====` are not used by the following phase and are present only for information. They state which SR variant was used, the number of PT-CT pairs, the key index, which corresponds to the file name, e.g., `0.in`, the secret key, and whether the Boolean polynomials are included. The first line after `=====` specifies the number of variables, followed by one line per variable name. Finally, the file contains the number of polynomials, followed by one line per polynomial. As this example was generated with Boolean polynomials included, the file contains 48 polynomials: 32 derived from the small-scale AES, and 16 Boolean polynomials. In other words, enabling the `"boolean"` flag results in the inclusion of $m = rce$ polynomials, one for each variable. We omitted parts of the file for brevity using `<TRUNCATED>`.

■ **Code listing 5.2** Generated `.in` file example

```

1 SR(1,2,2,4)
2 pairs:2
3 key_number:0
4 1101011110000011
5 boolean:true
6 =====
7 16
8 k000
9 k001
10 k002
11 k003
12 k010
13 <TRUNCATED>
14 k033
15 48
16 k000*k001*k003+k000*k001+k000*k002*k003+<TRUNCATED>+k032
17 k000*k001*k002+k000*k001+k000*k002*k003+<TRUNCATED>+k033
18 <TRUNCATED>
19 k010*k011*k012+k010*k011*k013+<TRUNCATED>+1
20 k000*k000+k000
21 <TRUNCATED>
22 k033*k033+k033

```

5.3 Benchmark Runner

Once again, we start by defining three terms:

Benchmark is a named combination of an input and implementations to be run on it.

Benchmark suite is a list of benchmarks. These are stored in the `benchmarks-spec.json` file.

Benchmark run is one measured run of one implementation on one input instance.

The rest of this section describes the main components of the benchmark runner and how they interact during benchmark execution.

5.3.1 Benchmark Suite Specification

We explain the benchmark suite using an example of `benchmarks-spec.json` in Code listing 5.3. It defines two benchmarks, called *pilot* and *another-pilot*. Both of them use the input *pilot* from `inputs-spec.json`, which shows that multiple benchmarks can use the same input, and that the benchmark and the input name can match, which can improve readability (the *main* input is not used in this example). The *pilot* benchmark will run all 64 input instances from the *pilot* input on two implementations, resulting in $2 \cdot 64 = 128$ benchmark runs. Each run will be given 10 minutes to complete, and at any moment in time there will be only one active benchmark run, thanks to **"parallelization": 1**.

The *another-pilot* benchmark differs from *pilot* in two ways. First, it uses the **"input-globs"** key. This key is optional and if it is present, it defines a subset of the input's instances, as they are stored in the directory structure described in Section 5.2. The syntax is passed directly to the `Path.glob()` function [98], which uses the Pattern language [99]. In short, `*` matches zero or more characters, `[abc]` matches exactly one of the characters written inside the square brackets, and `?` matches exactly one character. All of these work only on one path segment, i.e., they do not match the path delimiter `/`, which must be typed out explicitly. Therefore, all of these globs must contain exactly one `/`. The globs from the example correspond to:

"SR_1_*/[02].in" from all SR variants with $n = 1$ with PT-CT pair counts both 2 and 8 take input instances `0.in` and `2.in`.

"SR_2_2_2_4:pairs_2/0.in" take exactly this input instance.

"SR_?_4_*:pairs_2/[23].in" from all SR variants with $r = 4$ and 2 PT-CT pairs take input instances `2.in` and `3.in`.

■ Code listing 5.3 benchmarks-spec.json example

```
1  [
2    {
3      "name": "pilot",
4      "input-name": "pilot",
5      "timeout-min": 10,
6      "groups": [
7        {
8          "parallelization": 1,
9          "implementations": ["openf4", "macaulay2:gb"]
10       }
11     ]
12   },
13   {
14     "name": "another-pilot",
15     "input-name": "pilot",
16     "timeout-min": 20,
17     "input-globs": [
18       "SR_1_*/[02].in",
19       "SR_2_2_2_4:pairs_2/0.in",
20       "SR_?_4_*:pairs_2/[23].in",
21       "SR_1_4_2_8:pairs_8/*.in"
22     ],
23     "groups": [
24       {
25         "parallelization": 4,
26         "implementations": [
27           "singular:groebner",
28           "singular:std",
29           "singular:slimgb",
30           "singular:sba",
31           "singular:stdhilb"
32         ]
33       },
34       {
35         "parallelization": 2,
36         "implementations": ["magma:GroebnerBasis", "openf4"]
37       }
38     ]
39   }
40 ]
```

"SR_1_4_2_8:pairs_8/*.in" take all input instances from SR(1, 4, 2, 8) with 8 PT-CT pairs.

If multiple glob patterns overlap, the duplicate instances are used only once, which allows combining more general and more specific glob patterns without worrying about their intersection. The globbing example presented is synthetic and would probably never be used in real-life usage of this framework. However, it demonstrates the versatility of this globbing approach.

The second difference is that *another-pilot* splits implementations into two groups, each with a different parallelization value. The parallelization parameter defines the maximum number of concurrently running benchmark runs of the implementations in that group. In the example, this means that the Singular implementations can run with at most three other benchmark runs at a time, and that Magma and openf4 can both run concurrently with at most one other benchmark run. This division into groups does not imply that only implementations from one group run at the same time. For instance, these rules allow for Magma and slimb to run simultaneously, as this satisfies the parallelization constraints from both groups. We go over this mechanism in more detail in Section 5.3.5.

5.3.2 Unified Interface Layer

We also introduce implementation *interfaces*. These are abstractions over the different implementations that allow the Benchmark runner to work with a unified interface. They come in the form of Python scripts with predefined parameters. If the functionality of an interface is the same across multiple implementations, it can be reused. We define three interface stages: *PRE*, *MAIN*, and *POST*. Each benchmark run executes the mandatory *MAIN* interface, with optional *PRE* and *POST* stages executed before and after it when defined. However, the runtime duration and memory consumption are measured only for the *MAIN* interface.

5.3.2.1 The PRE Interface

The purpose of the *PRE* interface is to convert the input instance (the *.in* file) into a format suitable for the concrete implementation. The motivation for this interface is to prepare the data in a suitable format before we measure the implementation.

The *PRE* interface scripts take three arguments: `<IMPLEMENTATION>`, `<INPUT>`, and `<MAIN_INPUT>`. The first argument is the implementation name, e.g., `giac:gbasis:plex`. The second is the path to the *.in* file, and the third is the path to a *main.in* file. The interface loads the *.in* file, converts it to the desired format, and writes it to the *main.in* file. We note that the input instance *.in* files and the resulting *main.in* file have, in general, completely different structures, despite them sharing the same suffix.

This interface is useful for the implementations that have a predefined input format, e.g., `msolve` from Section 4.6. Other implementations do not have their own format, mostly those that come in the form of a library where we had to make our own program, e.g., `openf4` from Section 4.10. For these cases, the parsing has to be implemented manually in the program itself, so it does not make sense to use the PRE interface. For this reason, the PRE interface is optional, and if it is omitted, the Benchmark runner simply copies the input instance `.in` file to the `main.in` file without modifying it.

One exception to this usage of the PRE interface is the interface for `m4gb`. As we mentioned in Section 4.8, the first invocation of its `solver.sh` compiles the binary. Because this occurs only once for a given number of variables, we trigger the compilation in the PRE interface as well, so that compilation time is not included in the runtime statistics of `m4gb`. The measured MAIN invocation may still perform an up-to-date build check.

5.3.2.2 The MAIN Interface

The MAIN interface calls the implementation on the input instance and waits for its completion. This is the part that is actually being benchmarked. Its arguments are `<IMPLEMENTATION>` `<MAIN_INPUT>` `<MAIN_OUTPUT>`. The first argument is the same as before. The second one is the path to the `main.in` file generated by the PRE interface. The last argument is a path to a `main.out` file where the output is to be stored.

This interface specifies the implementation arguments, prepares its environment, provides the `main.in` file either as a command line argument or through standard input, and captures the result in `main.out` either via a command line argument or from standard output. It also returns the implementation's return code to the Benchmark runner. This interface is the only mandatory one.

5.3.2.3 The POST Interface

The POST interface can be seen as an inverse to the PRE interface. While the PRE interface converts the unified input instance format to the format specific to the implementations, the POST interface takes the result of the instance invocation and converts it back to a unified format. The benefit of this is that all results then share the same structure and can be easily checked just by looking at them. It also provides room for further processing without having to deal with the details of the formats used by the implementations.

The arguments are `<IMPLEMENTATION>` `<MAIN_OUTPUT>` `<RESULT>`. The first one is again the name of the implementation. The second one is the path to the `main.out` file generated by the MAIN interface. The last argument is the path to a `result.txt`. This is the file where the converted output from `main.out` is stored. The format is the resulting polynomials, one on each line. The information about the variable names is absent, as it is generally not

present in the `main.out` generated by the implementations. An example of a `result.txt` file is available in Code listing 5.4.

■ **Code listing 5.4** Example of `result.txt`

```

1 k000+k001+k003+k031+k032+k033+1
2 k001
3 k002
4 k003+k033+1
5 k010+k012+k013+k021
6 k011+k012+k013
7 k012+k023+1
8 k013+k023+1
9 k020+k021
10 k021+k023+1
11 k022+k023
12 k023+k030+k033+1
13 k030+k033
14 k031+k033
15 k032+k033+1
16 k033+1

```

5.3.3 Lifecycle of a Benchmark Run

This section describes the execution of a single benchmark run. The first thing that happens is the creation of the output folder for this benchmark run. It is in the following location: `<outputs-folder>/<benchmark>/SR_<n>_<r>_<c>_<e>:pairs_<pair-count>/<key-index>/<implementation>/`, e.g., `./outputs/another-pilot/SR_2_2_2_4:pairs_2/0/m4gb/`. We call it the Run directory. Then, a `run.log` file is opened in this directory. It will contain all the runtime data from this benchmark run, and is the most important benchmark file.

Next, the PRE interface is executed. The resulting `main.in` file is created in the Run directory. If the PRE interface does not exist, the input instance file is simply copied to `main.in`. After that, the MAIN interface is invoked, which runs the implementation itself. This invocation is given the time defined in the benchmark (see Section 5.3.1). It is also limited in terms of memory consumption, which will be explained in greater detail in Section 5.3.4.

The MAIN interface can finish in one of three ways:

FINISHED means the implementation completed the computation in the time budget it was given.

TERMINATED happens when the implementation did not finish in the time window and had to be terminated.

FAILED means the implementation finished with a non-zero return code. This can mean multiple things, such as an exception being raised, a segmentation fault occurring, or the implementation using too much memory and being killed by the system.

If the MAIN interface completed with the status **FINISHED**, then the `main.out` file is written to the Run directory. Then the POST interface is invoked, transforming it into the `result.txt` file, also in the Run directory.

If any of the interfaces fail, the remaining interfaces are not called. This is mainly relevant for the MAIN interface, as PRE and POST usually do not have an opportunity to fail. Also, as PRE and POST interfaces are orders of magnitude faster and less memory-intensive than the MAIN interface, we did not implement a timeout or memory limit for them.

No matter how the benchmark run ends, the `run.log` file is filled with data. This is the main output of the Benchmark runner. We now describe its format. An example of a **FINISHED** run is available in Code listing 5.5.

The lines have the format `===== <KEY> =====` or `===== <KEY>: <VALUE> =====`. Any subsequent lines that do not start with `=====` after a line of the first format become its value. If there is no such line, its value is empty. Lines without `=====` after a line of the second format are not permitted. In this way, this file creates a key-value map from strings to strings. Simultaneously, this format is easy to produce and is also human-readable.

The example log contains the following information:

1. Environment information (more details in Section 5.3.4).
2. Statistics about the PRE interface, together with its outputs.
3. Statistics about the MAIN interface, together with its outputs.
4. Collected runtime information about the MAIN interface, which describes the implementation execution. It contains information such as execution time, memory consumption, and CPU pressure. It also provides data from the systemd journal, which can indicate whether the process was killed because it ran out of memory (more details in Section 5.3.4).
5. Statistics about the POST interface, together with its outputs.

A schema of the complete benchmark run is shown in Figure 5.2.

5.3.4 Resource Control Using `cgroups v2` and `systemd-run`

The two main metrics we are interested in are execution duration and memory consumption. Getting the execution duration is trivial. We essentially need to

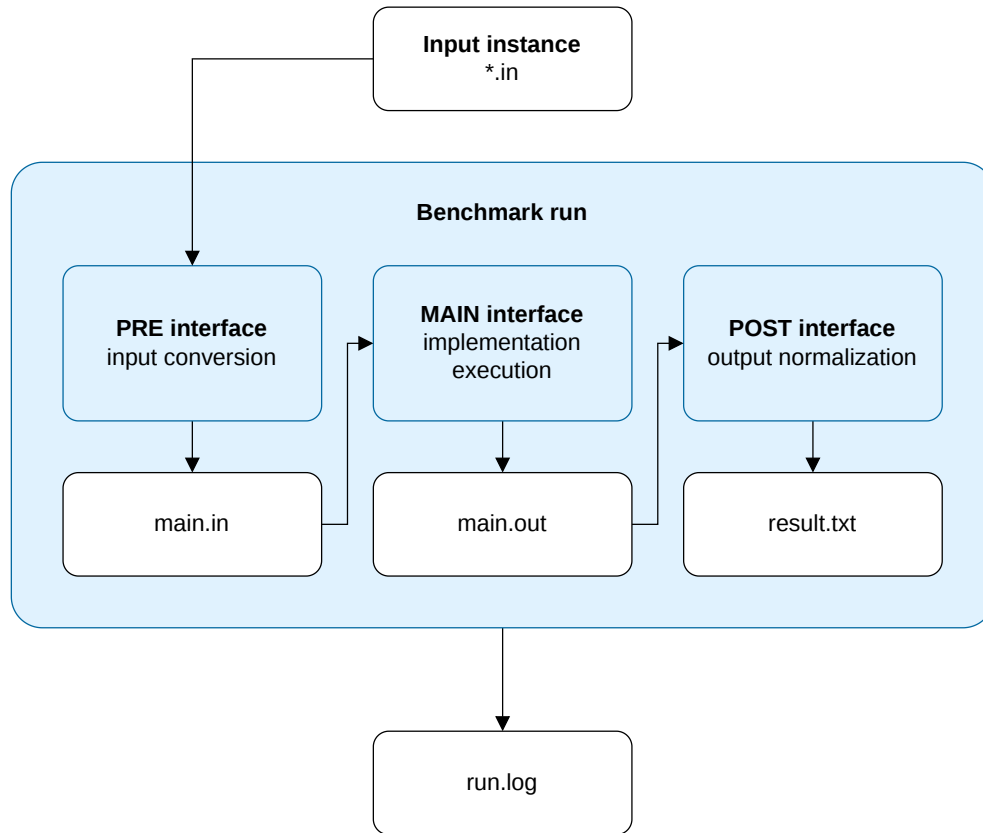
■ Code listing 5.5 run.log example

```

1  ===== BENCHMARK VERSION: 33 =====
2  ===== BENCHMARK SLICE: benchmark-workers.slice =====
3  ===== UNIT NAME: 1f7e5061-c3d8-4e29-a19f-36d50b5bfe1e =====
4  ===== RUN DIRECTORY: outputs/pilot54/SR_2_2_2_4:pairs_2/2/m4gb =====
5  ===== MAIN INPUT PATH: outputs/pilot54/SR_2_2_2_4:pairs_2/2/m4gb/main.in =====
6  ===== MAIN OUTPUT PATH: outputs/pilot54/SR_2_2_2_4:pairs_2/2/m4gb/main.out =====
7  ===== RESULT PATH: outputs/pilot54/SR_2_2_2_4:pairs_2/2/m4gb/result.txt =====
8  ===== PRE: python -m 'interfaces.pre.m4gb' 'm4gb' '.../TRUNCATED' =====
9  ===== PRE START TIME: 2026-04-22 10:40:41.490 =====
10 ===== PRE END TIME: 2026-04-22 10:40:41.783 =====
11 ===== PRE EXECUTION TIME: 0:00:00.292342 =====
12 ===== PRE STDOUT =====
13 make: Entering directory '/scratch_ssd/holypav2/implementations/src/m4gb'
14 Executable: bin/solver_m4gb_n16_gf2
15 make: Leaving directory '/scratch_ssd/holypav2/implementations/src/m4gb'
16
17 ===== PRE STDERR =====
18
19 ===== MAIN: python -m 'interfaces.main.m4gb' 'm4gb' 'outputs/pilot54/.../TRUNCATED' =====
20 ===== MAIN FINISHED =====
21 ===== MAIN STDOUT =====
22 Inputfile: /scratch_ssd/holypav2/diplomka/experiments/benchmark/outputs/.../TRUNCATED>
23 Fieldsize: 2
24 Variables: 16
25 Solver : bin/solver_m4gb_n16_gf2
26 Options : --loglevel 0 -o /scratch_ssd/holypav2/diplomka/experiments/.../TRUNCATED>
27
28 === Building solver binary ===
29 make: 'bin/solver_m4gb_n16_gf2' is up to date.
30
31 === Running solver binary ===
32 bin/solver_m4gb_n16_gf2 --loglevel 0 -o /scratch_ssd/holypav2/diplomka/.../TRUNCATED>
33 Total CPU time : 28.1733 s
34 Peak memory : 3189 MiB
35 Resource index : 89844.8 s*MiB
36
37 ===== MAIN STDERR =====
38 22.46user 5.91system 0:21.96elapsed 129%CPU (0avgtext+0avgdata 3266060maxresident)k
39 0inputs+8outputs (0major+1235737minor)pagefaults 0swaps
40
41 ===== CONSUMED MEMORY: 3373137920 =====
42 ===== CPU USAGE NSEC: 28432532000 =====
43 ===== CPU PRESSURE SOME AVG10 PEAK: 0.00 at 2026-04-22 10:40:42.786 =====
44 ===== CPU PRESSURE SOME AVG60 PEAK: 0.00 at 2026-04-22 10:40:42.786 =====
45 ===== CPU PRESSURE SOME AVG300 PEAK: 0.00 at 2026-04-22 10:40:42.786 =====
46 ===== CPU PRESSURE SOME TOTAL USEC: 10842 =====
47 ===== CPU PRESSURE FULL AVG10 PEAK: 0.00 at 2026-04-22 10:40:42.786 =====
48 ===== CPU PRESSURE FULL AVG60 PEAK: 0.00 at 2026-04-22 10:40:42.786 =====
49 ===== CPU PRESSURE FULL AVG300 PEAK: 0.00 at 2026-04-22 10:40:42.786 =====
50 ===== CPU PRESSURE FULL TOTAL USEC: 10780 =====
51 ===== MAIN START TIME: 2026-04-22 10:40:41.784 =====
52 ===== MAIN END TIME: 2026-04-22 10:41:03.840 =====
53 ===== MAIN EXECUTION TIME: 0:00:22.056067 =====
54 ===== JOURNALCTL =====
55 2026-04-22T10:40:41+02:00 gpu2 systemd[700835]: Started 1f7e5061-c3d8-4e29-.../TRUNCATED>
56 2026-04-22T10:41:03+02:00 gpu2 systemd[700835]: 1f7e5061-c3d8-4e29-a19f-.../TRUNCATED>
57 ===== POST: python -m 'interfaces.post.split-lines' 'm4gb' 'outputs/.../TRUNCATED' =====
58 ===== POST START TIME: 2026-04-22 10:41:04.851 =====
59 ===== POST END TIME: 2026-04-22 10:41:04.888 =====
60 ===== POST EXECUTION TIME: 0:00:00.036429 =====
61 ===== POST STDOUT =====
62
63 ===== POST STDERR =====
64
65 ===== FINISHED =====

```

capture the current time, run the implementation, and then capture the time again. The difference is the duration. Alternatively, we can use tools such as the `time` Linux command [100].



■ **Figure 5.2** Overview of a benchmark run

The problem comes with capturing the memory, and for two reasons. The first reason is that we are interested in the peak memory consumption of the whole invocation. The second reason is that while many implementations execute only as one process, some can spawn subprocesses. Without examining the source code of all the implementations, one cannot guarantee that all implementations use strictly one process. And even if it were the case for the implementations we used, one would have to check every new implementation to determine whether it is also strictly single-process. Therefore, we need to take into account the memory of the process and all of its descendants. In theory, we need to sum the memory consumption of the process and all of its descendants at every moment. We then take the maximum of these sums across the whole invocation. The resulting value is the peak memory consumption we are looking for.

Throughout the development of this framework we tried several approaches to solve this.

5.3.4.1 Approach 1 – `time -v`

The first approach was to run the `time` utility with the `-v` switch. We emphasize that this is the command, not the shell built-in of the same name. To execute the command, we usually need to specify the whole path, e.g., `/usr/bin/time`. The built-in of `bash` does not have the `-v` switch. [100]

According to the manual pages, this makes it “Give very verbose output about all the program knows about”. In practice, it prints multiple lines containing various information about the program upon its completion. One of the lines is `Maximum resident set size (kbytes): <VALUE>`. This does indeed represent the maximum memory used by the process [100].

The problem arises when the process has descendants. It does not ignore them, but it seems to measure their memory consumption separately and report the maximum from that. We demonstrate this on the following example. We call `time -v` on a Python process. This process first allocates some memory by creating a list of a selected size. It then creates a subprocess that does the same. This can be done by calling `/usr/bin/time -v python -c 'a = [123] * <SIZE1>; from subprocess import run; run(["python", "-c", "b = [123] * <SIZE2>"])'`, where we substitute sizes for `<SIZE1>` and `<SIZE2>`, controlling the list sizes of the parent and of the child, respectively. We give the results of all four combinations of small/large lists in Table 5.1.

■ **Table 5.1** Test of `time -v` on two processes

<SIZE1>	<SIZE2>	Maximum resident set size (kbytes)
1	1	13120
1	100000000	792496
100000000	1	794240
100000000	100000000	794368

From the first three lines of the table, it seems to work as desired. It reports a large memory usage when there is a large array. It also discovers the large array in the main process as well as in the child. The problem appears in the last line, where the memory consumption should be roughly twice what it is, as we allocate two such arrays. Instead, it reports the maximum of the memory usage of the processes separately, not combined. This behavior makes it unsuitable for our use case.

5.3.4.2 Approach 2 – Process Groups

Next, we focused on *process groups*. Every process belongs to a process group (in other words, it has its own *pgid*), and when it creates a subprocess, the subprocess has the same *pgid* by default. We can also change the *pgid* of a

process manually. This way, we can make the MAIN interface start the implementation in a new process group. We can then periodically (e.g., every second) read the memory usage of all processes, filter out processes with different pids, and sum the memory usage of the remaining ones. At the end, we would return the maximum observed value. [101]

The benefit of this approach was also in the problem of implementing the termination if the implementation exceeded the timeout limit. Sending a simple **SIGTERM** signal via `kill <PID>` to the implementation process is not sufficient, as it does not terminate its descendants. Their parent will be set to the process with PID 1 and they will keep running. Instead, we can use the `killpg` function, which sends the **TERM** signal to the whole process group, terminating them all. [102]

This worked almost always, except for Macaulay2. Upon deeper inspection, we discovered that Macaulay2 used process groups itself. This broke the assumption that all descendants would belong to the same process group and rendered this approach unreliable, and thus unusable.

5.3.4.3 Approach 3 – Traversing the Process Tree

The third approach built on the previous one, periodically polling the memory usage of processes, summing them, and finally taking a maximum. The only difference was how we found out which processes were part of this benchmark run and which were not. Each time we queried the running processes, we constructed a directed acyclic graph (DAG). The processes were the vertices, and the edges were directed from parents to their children. We then ran breadth-first search (BFS) from the process launched by the MAIN interface. This way, we reliably collected the set of the process and all of its descendants.

The termination worked analogously – first, we constructed the set of the processes the same way by calling BFS, and then sent **SIGTERM** to all of them.

This approach still had its downsides:

1. Active polling is always worse than a single query at the end of the execution.
2. If the processes' memory usage spiked between two polls, this mechanism would not register it.
3. The process tree had to be traversed, which is slower than the previous approaches.

Ultimately, we had to decide on the polling period – too long, and we would miss spikes. Too short, and the process-tree polling and traversing would stop being negligible computation-wise. As mentioned previously, we are building on top of Python's `asyncio`, which is not meant to do CPU-intensive tasks. Despite all of these disadvantages, we settled for a period of one second and successfully used this implementation. We did not encounter any problems,

and it proved reliable. It would have been sufficient to settle for this implementation. Nevertheless, we discovered a completely different mechanism that removed all of the downsides and also simplified the implementation.

5.3.4.4 The Final Approach – cgroups v2 and systemd-run

The Linux kernel contains a mechanism called a *control group*, or *cgroup* for short. The original version was superseded by cgroup v2. This was a complete rewrite from v1, and its architecture is different. We focus only on cgroup v2 and may omit the version in the remainder of this text. [103]

A cgroup is a mechanism to organize processes hierarchically and distribute system resources along the hierarchy. cgroups form a tree structure, and every process belongs to exactly one cgroup. Furthermore, children of a process are put in the same cgroup as the parent. This behavior is similar to that of process groups. [103]

The key feature is that both resource limiting and resource accounting can be set on any part of the cgroup hierarchy. For instance, we can have three processes in one cgroup with a combined limit of 10 GiB of memory, and then set a separate limit for the first process to 300 MiB. We describe how we use the hierarchy later.

cgroups are accessed as a filesystem in the `/sys/fs/cgroup/` directory. The directory structure there corresponds to the cgroup hierarchy, and the user can modify the hierarchy by creating or deleting directories. The directories have files, which are used for limiting resources by writing to them and for querying information about the cgroup by reading from them. Which files are available is documented in [103], and it has evolved over the different kernel versions. All metrics we will present are available in Ubuntu 24.04.6 LTS running on kernel version 6.8.0-101-generic.

Instead of working with cgroups directly, we leveraged their implementation in *systemd*. *systemd* is a system and service manager for Linux operating systems. On many distributions it is the init system, i.e., the first process launched. Again, cgroups were adapted granularly. Everything we describe works on *systemd* version 255.4. [104]

While *systemd* is capable of numerous things, we utilize only a subset of them. The central one is **systemd-run**. It allows us to run a command in a *transient scope unit*. This means that **systemd-run** is still the parent of the command and finishes after the command finishes, but we can simultaneously use **systemctl** to query information about the command. [105, 106]

We illustrate the purpose of this mechanism with an example. Assume we want to launch some command, e.g., by the MAIN interface. We denote the command together with its arguments as `<COMMAND>`. Instead of launching it directly, we run the command from Code listing 5.6 instead. The arguments are as follows: [105]

--user runs it from the user *systemd* instance, i.e., not under root.

`--unit "<UNIT-NAME>"` assigns a chosen name. In our implementation, we use UUID4 (Universally Unique Identifier) [107] to make sure no two benchmark runs use the same unit name.

`--scope` runs in a transient scope unit.

`-p MemoryAccounting=yes` enables collection of memory-related information.

`-p CPUAccounting=yes` enables collection of CPU-related information.

`--quiet` suppresses extra informational output from `systemd-run`.

`bash -c "<COMMAND>"` executes the command.

■ Code listing 5.6 `systemd-run` invocation

```

1 systemd-run \
2   --user \
3   --unit "<UNIT-NAME>" \
4   --scope \
5   -p MemoryAccounting=yes \
6   -p CPUAccounting=yes \
7   --quiet \
8   bash -c "<COMMAND>"

```

Executing the command this way looks exactly as by calling `<COMMAND>` directly. The difference is that we can interact with the unit while it is running via `systemctl`. The first useful command is `systemctl stop --user "<UNIT-NAME>.scope"`. This kills the `<COMMAND>` together with all of its descendants. This way, we do not have to traverse the process tree anymore. [106]

The second command, which is the one that makes this approach conceptually superior to all the previous ones, is `systemctl show --user "<UNIT-NAME>.scope"`. This outputs all information `systemd` and `cgroup` know about the unit. One of the lines is `MemoryPeak`. According to the documentation, this is “The max memory usage recorded for the `cgroup` and its descendants since either the creation of the `cgroup` or the most recent reset for that FD”. This is exactly what we need. The value can also be read directly via `systemctl show --user "<UNIT-NAME>.scope" -p MemoryPeak --value`. [106, 103]

We explored the other information available in the `cgroups` aside from the memory peak. We were interested in two more fields, which we will describe shortly. The problem is that one of them was not exposed via `systemctl show`. Therefore, we resorted to manual reading from the `cgroup` filesystem. The `cgroup` path is conveniently exposed via `systemctl show --user <UNIT-NAME>.scope -p ControlGroup --value`, which outputs something like `/use`

`r.slice/user-<UID>.slice/user@<UID>.service/app.slice/<UNIT-NAME>.scope`. We then prepend `/sys/fs/cgroup` to it and end up with the directory of the cgroup for this unit. From there, we can read the three files we are interested in: [103]

`memory.peak` is what we previously got from `-p MemoryPeak`.

`cpu.stat` contains various information about CPU usage. We are interested in the first line, which is `usage_usec <VALUE>`. It tells us how much CPU time was used by the unit. This is different from the real execution time, as it calculates the combined CPU time across all cores. By dividing this time by the wall time, we get an average number of CPU cores used by the implementation, which is a valuable metric.

`cpu.pressure` displays the amount of time for which the unit was stalled on CPU, both as the total duration and as average percentages over the last 10, 60, and 300 seconds. If this value indicates that the CPU was not available, it suggests our benchmark is not correctly configured. This could happen if we simultaneously launched multiple implementations that heavily utilized threading. More details are available in [108].

The last problem was that the cgroup would disappear shortly after the unit finished. We examined three solutions:

1. Read from the cgroup as soon as the `systemd-run` finishes. This would probably work in practice, but there is a race condition between the read and the disappearance of the cgroup. We did not deem this robust.
2. Put the reading directly as part of the `<COMMAND>` executed by `systemd-run`. This eliminated the race condition, but did not work if the `<COMMAND>` got terminated or killed for using too much memory.
3. Read the values from the cgroup periodically from the Benchmark runner. While this might seem as though it would have the same disadvantages as the previous polling methods, this is not the case. For one, doing three read operations on the filesystem is faster than traversing the process tree. For another, the problem of missing spikes is eliminated. This is because `memory.peak`, `cpu.stat usage_usec`, and the `total` fields in `cpu.pressure` are non-decreasing, so for these counters we only care about the last value. The sampled `avg10`, `avg60`, and `avg300` pressure averages are not cumulative counters, so the runner records their maxima over the periodic samples instead. We configured the period to one second. While it is true that we may miss an update in the last second, the chance of missing a new memory peak is negligible, and being off by one second in both CPU metrics is also negligible, as we are benchmarking on the order of an hour.

Running the implementation via `systemd-run` also has the benefit that we can query system logs about it via `journalctl --user --unit "<UNIT-NAME>.scope"`. This can contain useful information, such as when the unit was terminated by the system for using too much memory. This was part of the `run.log` from Code listing 5.5, together with the values from `memory.peak` (`CONSUMED MEMORY:`), `cpu.stat` (`CPU USAGE NSEC:`), and `cpu.pressure` (`CPU PRESSURE *:`). [109]

Finally, we take advantage of the hierarchical nature of cgroups to set memory limits. We configure a new systemd slice with limits; the command is available in Code listing 5.7. This code creates a slice under the name `benchmark-workers.slice` in userspace. It then sets a memory limit to 700 GiB and a swap limit to 0 shared among all units that become part of it. We then start the slice by `systemctl --user start benchmark-workers.slice`. To make the `systemd-run` invocations part of this slice, we add the `--slice benchmark-workers.slice` option to the arguments presented in Code listing 5.6. Once the Benchmark runner completes all of its work, it cleans up the slice via `systemctl --user stop benchmark-workers.slice`. [106]

■ **Code listing 5.7** Configuring a systemd slice

```
1 systemctl \
2   --user \
3   set-property \
4   --runtime \
5   benchmark-workers.slice \
6   MemoryAccounting=yes \
7   MemoryMax=700G \
8   MemorySwapMax=0
```

Without the memory limitation enforced by the slice, the benchmark runs used to consume all available system memory (754 GiB), at which point the system killed something. In some cases, it killed the implementation that used the most memory, which is desirable. However, other times it decided to kill its parent, i.e., the Benchmark runner itself, which was unacceptable. Also, until we set the swap limit to 0, the behavior was that the memory-intensive implementation would use all 700 GiB of memory and no more, as the slice did not let it. However, it then started using more and more swap. Again, the system then decided to kill something, and sometimes it killed the Benchmark runner. Only once we set both limits was the behavior correct. Once the implementations collectively used up all the memory, they wanted to use the swap, but its limit is set to 0. Thanks to that, at that moment, the system killed something, but it killed something from this slice. This is because the system-wide memory and swap resources were not yet close to depletion.

There is no guarantee which of the processes from the slice will be killed, although based on our testing, it seems to be the one consuming the most memory. The important thing is that it will no longer be killing the Benchmark runner or something higher in the process hierarchy. Nevertheless, if some implementations use up all the provided memory, they should be run with parallelization 1 (as defined in Section 5.3.1). This is so that it is guaranteed that it is the memory-intensive implementation that gets killed and also so that it does not limit the other implementations.

For the sake of completeness, the Benchmark runner also queries the current memory usage and CPU pressure every ten seconds, but this time from the whole slice. It also gets the raw system memory usage by reading from `/proc/meminfo`. These values get logged for debug purposes.

To the best of our knowledge, there is no better resource-monitoring approach than using control groups, as they provide statistics captured directly by the kernel. We conclude that this solution is robust and very accurate.

5.3.5 Parallel Execution and Scheduling

We have to execute many benchmark runs, many of which will run until they time out. When running tens of implementations on multiple input instances, the duration of the whole measurement increases quickly. To mitigate this, we allow for parallel execution of benchmark runs. We need a design that allows both running multithreaded or memory-intensive implementations sequentially, and running singlethreaded implementations in parallel.

To accomplish this, we divide implementations into *groups*. Each group associates a list of implementations with *parallelization*. The parallelization value specifies at most how many implementations can run in parallel if at least one of them belongs to this group. Each benchmarked implementation must belong to exactly one group. An example of this mechanism appeared in Code listing 5.3. The *pilot* benchmark had only one group with two implementations, and all of the benchmark runs ran sequentially, because the parallelization value was 1. The *another-pilot* benchmark split the implementations into two groups, one with parallelization 4 and the other with parallelization 2.

This design allows for any number of groups. In practice, however, we usually used at most two: one with parallelization 1 for multithreaded or memory-heavy implementations, and some other parallelization for implementations that can safely operate in parallel without limiting the others. We used 16, but any other value could be used as well. If we knew exactly how many threads some implementation uses at most, we could divide the number of available threads by it and create a group with such a parallelization containing only this implementation.

We also had to implement a scheduler that respects these constraints. All benchmark runs are stored in a queue. The scheduler is then responsible for taking the benchmark runs from the queue and executing them. It keeps

the parallelizations of the implementations that are currently being run. It starts the next benchmark run if and only if the total number of currently running implementations would exceed neither the tracked parallelizations nor the parallelization of the next implementation.

The order of the benchmark runs in the queue matters as well. Assume we have 32 benchmark runs, each with parallelization 16 (group *A*), and 32 benchmark runs with parallelization set to 1 (group *B*). Also assume all of the benchmark runs will take exactly one hour. First, we put them in the queue interleaved, i.e., the runs in the queue come from the groups *A*, *B*, *A*, *B*, ..., *B*. The first run from *A* starts. It would allow another run to start in parallel. However, the next run is from *B*. As it has parallelization set to 1, the scheduler must wait until the first run finishes. It then starts the second run. It has to wait for its completion, as runs from *B* have to run sequentially. Then, it starts the third run, which belongs to *A*, but again has to wait before starting the fourth from *B*. This repeats until all runs are completed. For this queue, the runs are scheduled sequentially and take a total of $32 + 32 = 64$ hours.

To mitigate this, we put the runs in the queue sorted by decreasing parallelization, i.e., all runs from *A* followed by all runs from *B*. Now the scheduler launches the first run. When it tries to launch the second run, it does so, because that would make two runs execute in parallel, which violates neither of the two parallelization constraints, both of which are set to 16. This continues until 16 runs are active. Once one of them finishes, the next one is launched, until all 32 runs from *A* are either currently running or already completed. The scheduler then has to wait until all runs from *A* are finished, and then it launches all runs from *B* sequentially. This takes $\frac{32}{16} + 32 = 34$ hours, which is the best we can achieve.

If the group *B* had a greater parallelization value, for example 5, then once the last 5 runs from *A* are running and one finishes, the scheduler would launch the first run from *B*. In other words, the scheduler overlaps the groups whenever possible.

5.3.6 Invocation and CLI

We conclude the Benchmark runner section by giving a practical overview of its usage and command line parameters. The Benchmark runner is invoked with exactly one positional argument, the name of the benchmark to run, e.g., `./bench.py another-pilot`. It begins by reading the benchmark suite from `benchmarks-spec.json` and loading the selected benchmark. It prints `===== PROLOGUE =====`, followed by startup information. This information includes the version of the Benchmark runner, output file locations, cgroup memory limit, summary of which computations are already done and which need to be done, and a timeout-based upper estimate on the duration of this invocation.

It then prints `===== BENCHMARK RUN =====` and starts the scheduler that runs the interfaces, which finally start the implementations. It prints various details about the benchmark runs, such as the progress of the PRE/MAIN/POST interface stages and information from the scheduler. It also prints the system resource summary every ten seconds, as we described in Section 5.3.4.4. Once a run finishes successfully, times out, or is terminated, a status line summarizing the current run counts is printed in the format `[<SUCCESSFUL-RUNS>/<TIMED-OUT>/<FAILED>/<TOTAL>]`.

Once all benchmark runs are completed, the line `===== EPILOGUE =====` is printed. It is followed by information about the resource usage from the whole invocation (this is only for informative purposes, this information is not used anywhere), as well as the total runtime.

The Benchmark runner can be interrupted from the keyboard by pressing `Ctrl+C`. If that happens, the currently active benchmark runs are canceled and their incomplete run directories are removed. Upon the next invocation, only the incomplete benchmark runs are computed.

It also creates two log files. The first one is a mirror of the output written to the terminal, and is stored in `./logs/<BENCHMARK>.log`. The second is output from `journalctl` from all user processes during the runtime of the Benchmark runner. It is prepended by a header saying the start and end times, Benchmark runner version and benchmark name. It is stored in `./journal/<BENCHMARK>.log`. This is useful in the cases where the system kills something that we did not account for. But unless the Benchmark runner has a bug which we are not aware of, this should not happen and printing the journal is only defensive. However, one should be aware that the journal contains information about other processes as well, and should be treated with caution when sharing.

The Benchmark runner takes the following parameters:

- `--inputs-folder <directory>` is the path to the directory with generated input files. It defaults to `./input-gen/inputs`.
- `--outputs-folder <directory>` is the path to the directory where benchmark output files are stored. It defaults to `./outputs`.
- `--benchmarks-spec <file>` is the path to the `benchmarks-spec.json` benchmark suite file. It defaults to `./benchmarks-spec.json`.
- `--log-dir <directory>` is the path to the directory where benchmark log files are stored. It defaults to `./logs`.
- `--journal-dir <directory>` is the path to the directory where exported user journal files are stored. It defaults to `./journal`.

We designed the Benchmark runner to our use case, i.e., measuring Gröbner basis computation implementations. However, thanks to the expressive nature

of the benchmark suite, as well as the PRE/MAIN/POST interface modular architecture, it should be easy to use it for benchmarking any other type of software with little to no modification. This is demonstrated by the fact that it is capable of working with 21 different implementations, which have vastly different inputs, outputs, invocation methods, and range from binaries to scripts to C++ libraries.

5.4 Output Parser

The Output parser is the final phase of the benchmarking framework. Its purpose is to take all `run.log` files from a benchmark and convert them to a JSON file to simplify downstream processing. It can be invoked with one or more benchmark names, e.g., `./parse.py another-pilot`. For each benchmark, it reads the `run.log` files and stores the resulting JSON in `./parsed/<BENCHMARK>.json`.

The data from a specific benchmark run are accessed using the following indexing scheme: `root[<INPUT-INSTANCE>][<IMPLEMENTATION>][<KEY-INDEX>]`, e.g., `root["SR_2_2_4_4:pairs_16"]["magma:GroebnerBasis"][0]`. This points to an object with a "raw" key, which contains all the key-value pairs taken directly from `run.log`, described in Section 5.3.3. It then takes selected metrics and stores them under separate keys. These include the key index, status ("FINISHED"/"TERMINATED"/"FAILED"), return code of the implementation, Benchmark runner version, peak memory consumption, wall time, CPU usage time, the average number of CPU cores used, and CPU pressure. For details, see Section 5.3.4.4. An example of such a file is shown in Code listing 5.8.

In addition to the variable number of benchmark arguments, the Output parser accepts the following parameters:

`--outputs-folder <directory>` is the path to the directory with benchmark output files. It defaults to `../benchmark/outputs`.

`--parsed-folder <directory>` is the path to the directory where the parsed JSON files are stored. It defaults to `./parsed`.

■ **Code listing 5.8** Generated JSON from Output parser (truncated)

```

1 {
2   "SR_2_2_4_4:pairs_16": {
3     "magma:GroebnerBasis": [
4       {
5         "raw": {
6           "<TRUNCATED>": "",
7           "consumed_memory": "26171801600",
8           "cpu_usage_nsec": "201225349000",
9           "<TRUNCATED>": "",
10          "main_execution_time": "0:03:21.061556",
11          "<TRUNCATED>": "",
12          "finished": ""
13        },
14        "key_id": 0,
15        "status": "FINISHED",
16        "return_code": 0,
17        "benchmark_version_int": 33,
18        "consumed_memory_bytes": 26171801600,
19        "cpu_usage_seconds": 201.225349,
20        "execution_time_seconds": 201.061556,
21        "pre_execution_time_seconds": 10.706585,
22        "post_execution_time_seconds": 0.038333,
23        "cpu_pressure_some_seconds": 0.019141,
24        "cpu_pressure_full_seconds": 0.016694,
25        "cpu_pressure_some_avg10": 0.0,
26        "cpu_pressure_some_avg60": 0.0,
27        "cpu_pressure_some_avg300": 0.0,
28        "cpu_pressure_full_avg10": 0.0,
29        "cpu_pressure_full_avg60": 0.0,
30        "cpu_pressure_full_avg300": 0.0,
31        "average_cpu_cores": 1.000814641064451
32      }
33    ]
34  }
35 }

```

Experimental Results

In this chapter we use the benchmarking framework to evaluate the implementations from Chapter 4. The performance of the implementations depends greatly on the input variant on which they are tested, i.e., not only on the small-scale AES variant, but on the PT-CT pair count as well. Although we explored numerous variants, many of them were either too easy and were successfully computed by (nearly) all implementations in seconds, or they were too difficult and (nearly) all implementations timed out or crashed.

We selected four input variants for this work. The first one is derived from a trivial small-scale AES variant. Its purpose is to verify whether the implementations operate correctly on a basic input. The remaining three are based on polynomial systems of increasing difficulty, all of which use small-scale AES variants with 2 rounds, which greatly increases the complexity compared to the variant with only 1 round. The four input variants are as follows:

Pilot experiment using SR(1,2,2,4) with 2 PT-CT pairs.

Easy experiment using SR(2, 2, 2, 4) with 8 PT-CT pairs.

Medium experiment using $\text{SR}(2, 2, 2, 4)$ with 2 PT-CT pairs.

Hard experiment using SR(2, 2, 4, 4) with 8 PT-CT pairs.

Adding more PT-CT pairs also adds equations that share the same unknown key, making the system more overdefined. This can make Gröbner basis computation easier for some implementations, even though the system contains more polynomials. We therefore use 8 PT-CT pairs in two roles: for SR(2, 2, 2, 4), it gives an easier overdefined two-round variant used as the first challenging experiment, while the 2-pair version gives a harder, less-overdefined Medium experiment. For the larger SR(2, 2, 4, 4) variant, the increased state size makes the input more difficult, so we also use 8 pairs to increase the chance that at least some implementations can solve it.

Furthermore, we have performed all four experiments twice – with and without the Boolean polynomials, resulting in a total of eight experiments. Even though the version with the Boolean polynomials is more relevant, we include both, as some of the implementations performed better without the added polynomials.

As the runtime of an implementation can differ between instances of the same input variant, we tested ten instances generated from ten keys for each of the variants, resulting in a total of 80 input instances. The input specification of the first experiment without the Boolean polynomials is in Code listing 6.1. Input specifications for the other experiments are analogous. The timeout was configured to 1 hour for all experiments. The benchmark specification of the first experiment without the Boolean polynomials is in Code listing 6.2. The remaining benchmark specifications use the same structure, but differ in the selected input name, the evaluated implementations, and the parallelization values. We chose the parallelization values based on our CPU and memory usage observations to minimize the total runtime without causing resource starvation. All input and benchmark specifications are available in the attached files.

■ **Code listing 6.1** Pilot experiment input specification without Boolean polynomials

```

1 {
2   "name": "pilot",
3   "keys": 10,
4   "boolean": false,
5   "variants": [
6     {
7       "pairs_per_key": [2],
8       "SRs": [[1, 2, 2, 4]]
9     }
10  ]
11 }
```

In the summary tables, OK, TO, and Fail denote the numbers of successful, timed out, and failed runs, respectively. Time, memory, and CPU are reported as arithmetic means over successful runs only. Time is the wall-clock runtime in seconds, memory is the peak memory consumption in mebibytes, and CPU is the average number of CPU cores used during the run. Peak memory consumption and CPU usage are measured as described in Section 5.3.4.4. The CPU pressure reports from all runs confirmed that none of the runs were starved of CPU resources.

We now present the results of these experiments, starting with the version without the Boolean polynomials.

■ **Code listing 6.2** Pilot experiment benchmark specification without Boolean polynomials

```

1 {
2   "name": "pilot",
3   "input-name": "pilot",
4   "timeout-min": 60,
5   "groups": [
6     {
7       "parallelization": 16,
8       "implementations": [
9         "singular:groebner", "singular:std",
10        "singular:slimgb", "singular:stdhilb",
11        "singular:sba", "sage:buchberger",
12        "sage:buchberger2", "sage:d_basis",
13        "brial", "cocoalib"
14      ]
15    },
16    {
17      "parallelization": 1,
18      "implementations": [
19        "giac:gbasis:plex", "giac:gbasis:revlex",
20        "msolve", "magma:GroebnerBasis",
21        "macaulay2:gb", "macaulay2:f4",
22        "macaulay2:mgb", "macaulay2:booleangb",
23        "m4gb", "groebner.jl", "openf4"
24      ]
25    }
26  ]
27 }
```

6.1 Experiments Without Boolean Polynomials

This section presents a baseline set of experiments without the additional Boolean polynomials $x_i^2 + x_i$. The purpose is to compare how the same implementations behave when these extra polynomials are omitted from the generated input. All implementation settings are the same as in Chapter 4 and are reused in both experiment sets; in particular, implementations configured to use Boolean polynomial rings still use them here. Thus, “without Boolean polynomials” only means that the generated input does not include the additional polynomials $x_i^2 + x_i$.

6.1.1 Pilot Experiment Without Boolean Polynomials

The first experiment uses the trivial input variant of $\text{SR}(1, 2, 2, 4)$ with 2 PT-CT pairs and without the Boolean polynomials. Its purpose is to verify that the implementations are capable of at least some Gröbner basis computation. We present the results of this experiment in Table 6.1 and in Figures 6.1 and 6.2.

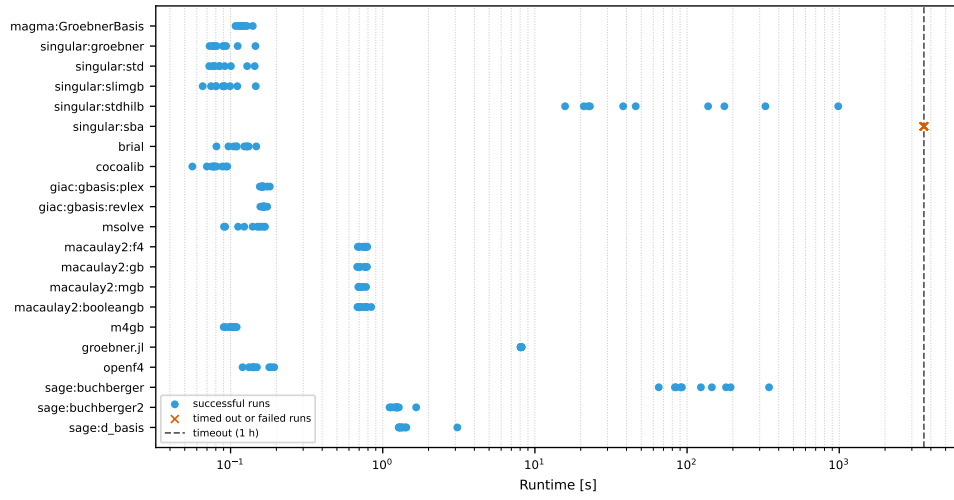
■ **Table 6.1** Summary of the Pilot experiment without Boolean polynomials

Implementation	OK	TO	Fail	Time [s]	Memory [MiB]	CPU
magma:GroebnerBasis	10	0	0	0.12	2.22	0.00
singular:groebner	10	0	0	0.09	0.00	0.00
singular:std	10	0	0	0.09	0.71	0.00
singular:slimgb	10	0	0	0.09	0.00	0.00
singular:stdhilb	10	0	0	179	50.1	0.98
singular:sba	0	10	0	–	–	–
brial	10	0	0	0.12	3.39	0.15
cocoalib	10	0	0	0.08	0.67	0.07
giac:gbasis:plex	10	0	0	0.16	0.00	0.00
giac:gbasis:revlex	10	0	0	0.17	1.71	0.00
msolve	10	0	0	0.14	4.12	0.90
macaulay2:f4	10	0	0	0.74	18.0	0.45
macaulay2:gb	10	0	0	0.73	8.72	0.00
macaulay2:mgb	10	0	0	0.72	8.80	0.51
macaulay2:booleangb	10	0	0	0.75	0.00	0.00
m4gb	10	0	0	0.10	9.57	0.00
groebner.jl	10	0	0	8.11	393	2.14
openf4	10	0	0	0.15	8.36	2.08
sage:buchberger	10	0	0	140	120	1.02
sage:buchberger2	10	0	0	1.25	118	2.92
sage:d_basis	10	0	0	1.51	118	2.54

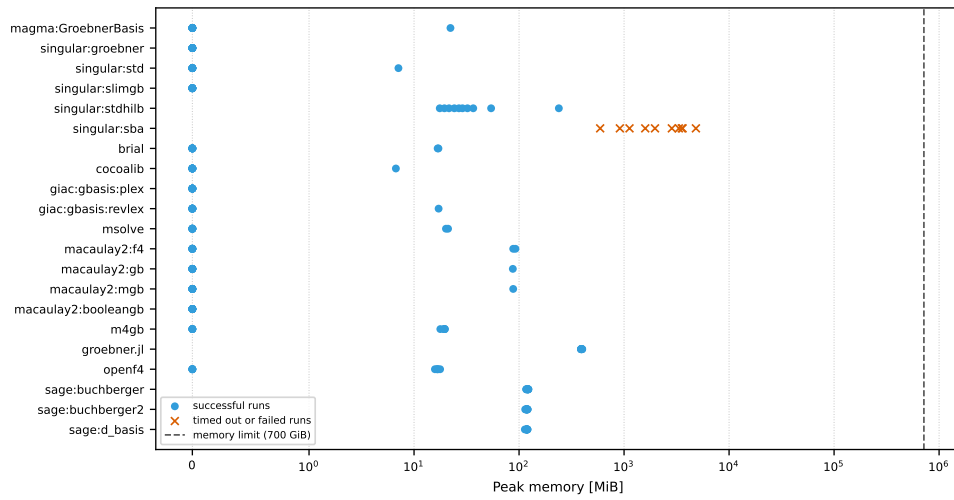
Time, memory, and CPU averages are computed over successful runs only.

`singular:sba` was the only implementation that did not recover a single key, while the other implementations recovered all keys successfully. Both `sage:buchberger` and `singular:stdhilb` took an average of about 2 to 3 minutes. However, the runtimes of `singular:stdhilb` were less consistent, reaching almost 1000 seconds on one of the keys. `groebner.jl` took around 8 seconds, and all the remaining implementations took on average 2 seconds or less. The fastest implementation in this experiment was `cocoalib`, with an average runtime of 0.08 seconds. None of the implementations failed.

CPU usage and memory peak are sampled every second, as described in Section 5.3.4.4. This explains the zero memory and CPU usage values. These values are not representative of the actual resource usage during such short



■ **Figure 6.1** Runtime of the Pilot experiment without Boolean polynomials. Each point represents one key.



■ **Figure 6.2** Peak memory consumption of the Pilot experiment without Boolean polynomials. Each point represents one key.

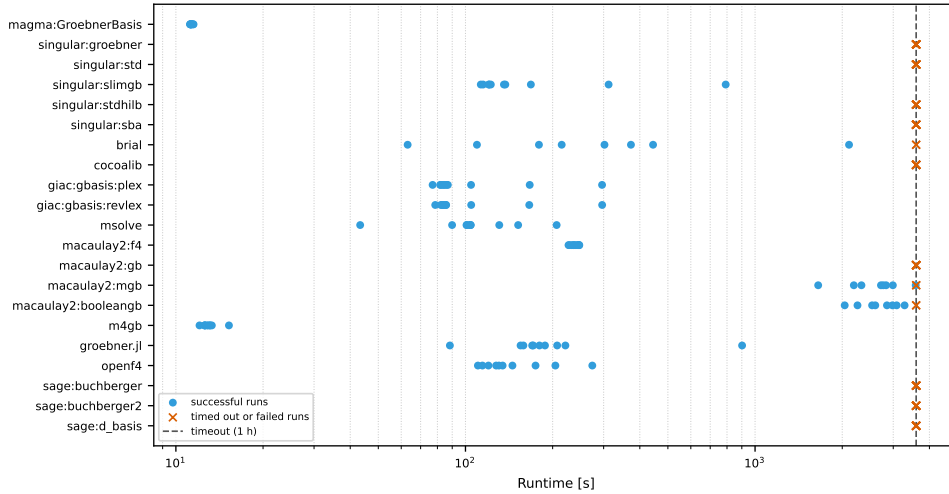
benchmark runs, and even if they were measured accurately, they would provide little information for such trivial inputs.

All implementations except `singular:sba` passed this test and mostly finished in a matter of seconds or less, as expected.

6.1.2 Easy Experiment Without Boolean Polynomials

This is the first challenging experiment, using the input variant of $\text{SR}(2, 2, 2, 4)$ with 8 PT-CT pairs and without Boolean polynomials. Its purpose is to rule out weak implementations and assess the overall behavior of the implementations.

The results shown in Table 6.2 and in Figures 6.3 and 6.4 summarize the behavior of the implementations. While none of the implementations failed, many of them timed out. This occurred for all Sage implementations. This was expected, as these were “toy” implementations not optimized for performance. All Singular implementations except `singular:slimgb` timed out as well. The last two implementations that were unable to find a solution within the time limit were `cocoalib` and `macaulay2:gb`. `brial` and the `macaulay2:mgb` and `macaulay2:booleangb` implementations timed out on exactly two instances. As shown in Figure 6.3, most of the benchmark runs of `brial` were much faster than the one-hour timeout. This was not the case for the two Macaulay2 implementations, as all runs took around half an hour or longer. All the remaining implementations recovered keys, with most of them having runtimes in the low hundreds of seconds. The only exceptions are `m4gb` and the reference implementation `magma:GroebnerBasis`, both of which took around 10 to 15 seconds on all runs. The fastest implementation in this experiment was `magma:GroebnerBasis`, with an average runtime of 11.3 seconds.

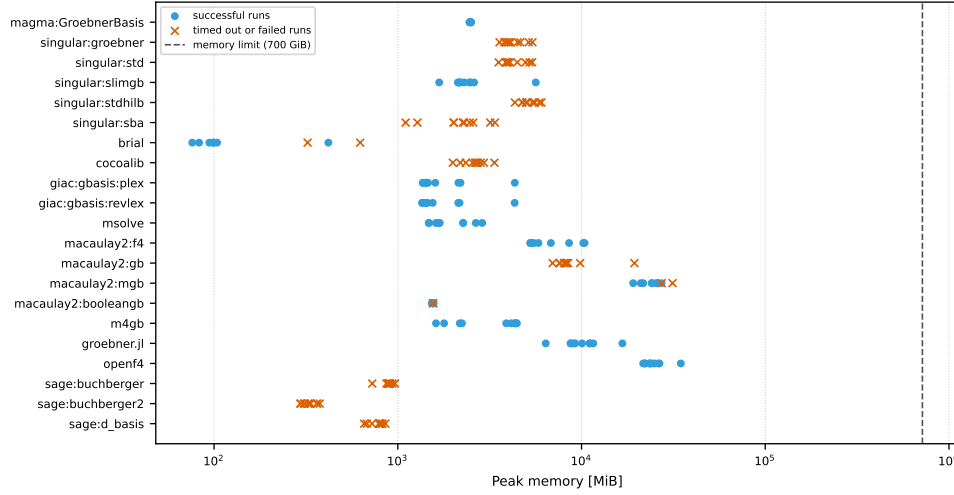


■ **Figure 6.3** Runtime of the Easy experiment without Boolean polynomials. Each point represents one key.

■ **Table 6.2** Summary of the Easy experiment without Boolean polynomials

Implementation	OK	TO	Fail	Time [s]	Memory [MiB]	CPU
magma:GroebnerBasis	10	0	0	11.3	2487	0.98
singular:groebner	0	10	0	–	–	–
singular:std	0	10	0	–	–	–
singular:slimgb	10	0	0	214	2581	1.00
singular:stdhilb	0	10	0	–	–	–
singular:sba	0	10	0	–	–	–
brial	8	2	0	475	134	1.00
cocoalib	0	10	0	–	–	–
giac:gbasis:plex	10	0	0	115	1874	2.86
giac:gbasis:revlex	10	0	0	115	1860	2.87
msolve	10	0	0	114	1967	32.9
macaulay2:f4	10	0	0	239	6855	3.90
macaulay2:gb	0	10	0	–	–	–
macaulay2:mgb	8	2	0	2635	23609	1.18
macaulay2:booleangb	8	2	0	2704	1553	1.17
m4gb	10	0	0	13.0	3340	1.12
groebner.jl	10	0	0	244	10267	15.5
openf4	10	0	0	154	24932	7.80
sage:buchberger	0	10	0	–	–	–
sage:buchberger2	0	10	0	–	–	–
sage:d_basis	0	10	0	–	–	–

Time, memory, and CPU averages are computed over successful runs only.



■ **Figure 6.4** Peak memory consumption of the Easy experiment without Boolean polynomials. Each point represents one key.

For the average peak memory consumption of the completed runs, most implementations used a few thousand mebibytes¹. `macaulay2:f4` used about

¹1MiB = 2²⁰B

7000 MiB and `groebner.jl` used around 10000 MiB. The maximum average memory consumption was around 25000 MiB for `macaulay2:mgb` and `openf4`.

The average number of CPU cores used by the successful implementations varied greatly. The largest use of multithreading was observed for `msolve`, which used an average of 32.9 cores. Unlike the memory metric, this is not a peak value, but an actual average. As the peak value is not captured, it is possible that certain implementations would benefit from an even higher core count. The next highest values were observed for `groebner.jl` and `openf4`, which used 15.5 and 7.8 cores, respectively. `macaulay2:f4` as well as both `giac:gbasis` implementations used about three to four CPU cores. `macaulay2:mgb`, `macaulay2:booleangb`, and `m4gb` all exhibited modest but measurable usage of multiple cores. The remaining implementations, i.e., `singular:slimgb`, `brial`, and `magma:GroebnerBasis`, all used one core. The value for Magma is slightly below 1. This is again because of the cgroup polling every second, which can lead to the reported average CPU usage being slightly lower than the real value. However, it cannot be higher than the real value. This means that even though `m4gb` also ran only for a few seconds, its average CPU usage was *at least* 1.12, indicating parallel CPU utilization during the run.

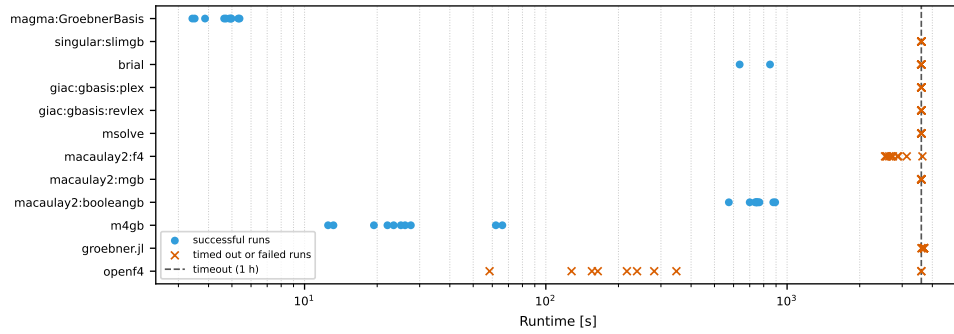
6.1.3 Medium Experiment Without Boolean Polynomials

This experiment uses $\text{SR}(2, 2, 2, 4)$ with 2 PT-CT pairs and without Boolean polynomials. For this experiment, the implementations from the Easy experiment that did not find a single key were removed. The results are shown in Table 6.3 and in Figures 6.5 and 6.6.

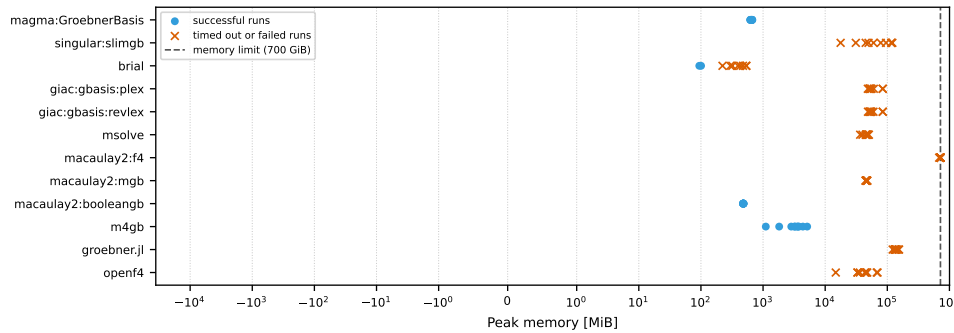
■ **Table 6.3** Summary of the Medium experiment without Boolean polynomials

Implementation	OK	TO	Fail	Time [s]	Memory [MiB]	CPU
<code>magma:GroebnerBasis</code>	10	0	0	4.56	650	0.74
<code>singular:slimgb</code>	0	10	0	—	—	—
<code>brial</code>	2	8	0	743	97.3	1.00
<code>giac:gbasis:plex</code>	0	10	0	—	—	—
<code>giac:gbasis:revlex</code>	0	10	0	—	—	—
<code>msolve</code>	0	10	0	—	—	—
<code>macaulay2:f4</code>	0	1	9	—	—	—
<code>macaulay2:mgb</code>	0	10	0	—	—	—
<code>macaulay2:booleangb</code>	10	0	0	755	480	1.14
<code>m4gb</code>	10	0	0	29.7	3273	1.30
<code>groebner.jl</code>	0	10	0	—	—	—
<code>openf4</code>	0	2	8	—	—	—

Time, memory, and CPU averages are computed over successful runs only.



■ **Figure 6.5** Runtime of the Medium experiment without Boolean polynomials. Each point represents one key.



■ **Figure 6.6** Peak memory consumption of the Medium experiment without Boolean polynomials. Each point represents one key.

This configuration filtered out many of the remaining implementations. Decreasing the pair count from 8 to 2 made many of the implementations that were previously successful fail to find solutions. Only **magma:GroebnerBasis**, **brial**, **m4gb**, and **macaulay2:booleangb** managed to find at least some solutions. The remaining implementations either timed out, consumed all memory and were killed (**macaulay2:f4**), or failed with nonzero return codes (**openf4**). **brial** managed to find two solutions. Those two invocations took on average 743 seconds. However, it timed out on all the remaining eight input instances. The other three implementations managed to find all ten keys. **macaulay2:booleangb** did so in a time similar to that of **brial**, while **m4gb** needed 29.7 seconds. The fastest implementation in this experiment was **magma:GroebnerBasis**, with an average runtime of 4.56 seconds. This is unexpected behavior, as decreasing the number of PT-CT pairs seems to make Magma faster. The same runtime decrease is present for **macaulay2:booleangb**, although its runtime is not nearly as impressive as that of Magma.

The memory consumption of `m4gb` is around 3200 MiB, while both `macaulay2:booleangb` and `magma:GroebnerBasis` use about 500–650 MiB, and `brial` is under 100 MiB for the two successful runs. The CPU usage is in line with the observations from the Easy experiment.

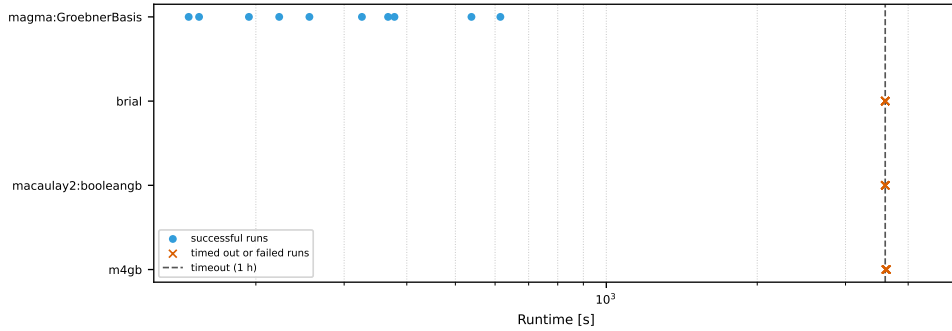
6.1.4 Hard Experiment Without Boolean Polynomials

This last experiment without Boolean polynomials uses $\text{SR}(2, 2, 4, 4)$ with 8 PT-CT pairs. All implementations that had not found a single solution in the Medium experiment were removed. This left only four implementations. We present the results in Table 6.4 and in Figures 6.7 and 6.8.

■ **Table 6.4** Summary of the Hard experiment without Boolean polynomials

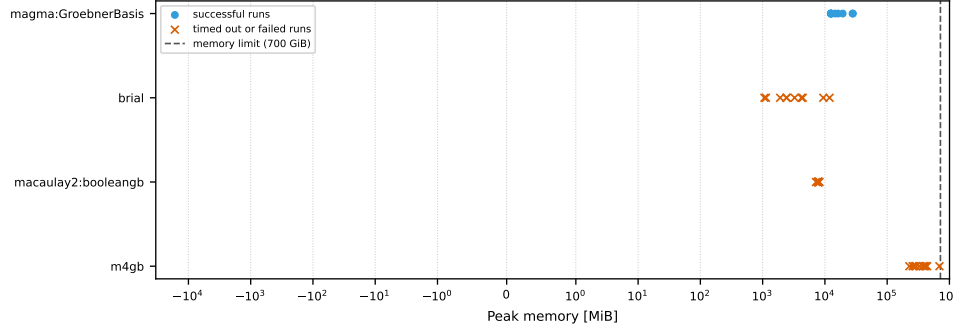
Implementation	OK	TO	Fail	Time [s]	Memory [MiB]	CPU
<code>magma:GroebnerBasis</code>	10	0	0	320	16858	1.00
<code>brial</code>	0	10	0	–	–	–
<code>macaulay2:booleangb</code>	0	10	0	–	–	–
<code>m4gb</code>	0	10	0	–	–	–

Time, memory, and CPU averages are computed over successful runs only.



■ **Figure 6.7** Runtime of the Hard experiment without Boolean polynomials. Each point represents one key.

This time, all three open-source implementations timed out. Magma found all ten solutions in about 5 minutes on average while consuming around 17 GiB of memory. The Hard experiment without the Boolean polynomials proved too difficult for the open-source implementations.



■ **Figure 6.8** Peak memory consumption of the Hard experiment without Boolean polynomials. Each point represents one key.

6.2 Experiments With Boolean Polynomials

This section presents the same four experiment variants, using the same small-scale AES parameters and PT-CT pair counts, this time with Boolean polynomials included. Each experiment again consists of ten generated input instances. The implementations are evaluated in the same way as before – the Pilot and Easy experiments evaluate all implementations, while the Medium and Hard experiments test only those that recovered at least one key in the previous experiment. The discussion focuses primarily on the differences and on the impact of including Boolean polynomials in the input files.

6.2.1 Pilot Experiment With Boolean Polynomials

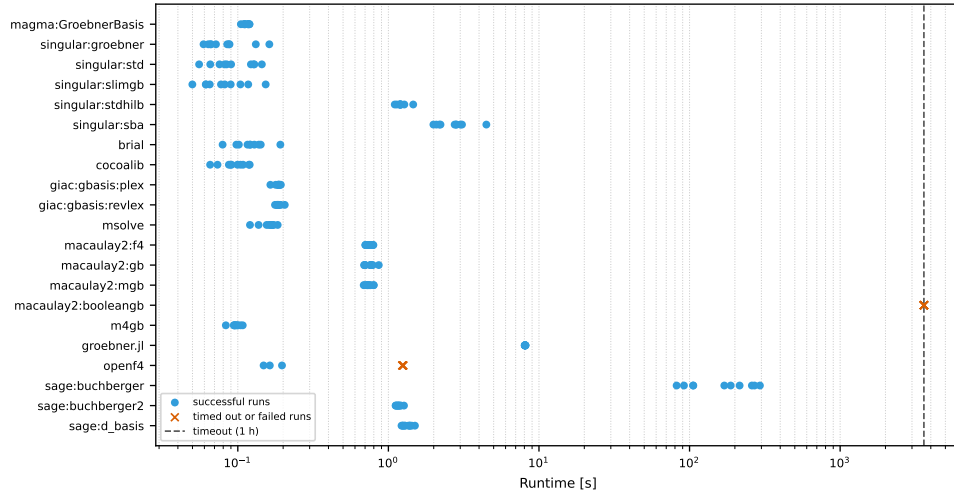
This is the first experiment with the Boolean polynomials, using the input variant of SR(1, 2, 2, 4) with 2 PT-CT pairs. The results are shown in Table 6.5 and in Figures 6.9 and 6.10.

Adding the extra polynomials led to mostly similar results, with a few exceptions. `singular:sba` previously did not find a single solution; now it found all of them. However, `macaulay2:booleangb` timed out on all instances. This is unexpected, as it was one of the best-performing implementations in the previous experiments. `openf4` managed to find three solutions in under a second, but failed with a nonzero return code in the remaining seven instances. The fastest implementations in this experiment were `singular:groebner` and `singular:slimgb`, both with an average runtime of 0.09 seconds. The runtime of `singular:stdhilb` decreased from 179 seconds to around 1 second. The remaining implementations performed similarly.

■ **Table 6.5** Summary of the Pilot experiment with Boolean polynomials

Implementation	OK	TO	Fail	Time [s]	Memory [MiB]	CPU
magma:GroebnerBasis	10	0	0	0.11	0.00	0.00
singular:groebner	10	0	0	0.09	0.72	0.06
singular:std	10	0	0	0.10	0.72	0.00
singular:slimgb	10	0	0	0.09	0.00	0.00
singular:stdhilb	10	0	0	1.22	8.04	0.81
singular:sba	10	0	0	2.74	144	0.81
brial	10	0	0	0.12	1.71	0.07
cocoalib	10	0	0	0.10	0.70	0.05
giac:gbasis:plex	10	0	0	0.18	0.00	0.00
giac:gbasis:revlex	10	0	0	0.19	0.00	0.00
msolve	10	0	0	0.16	0.00	0.00
macaulay2:f4	10	0	0	0.75	27.8	0.92
macaulay2:gb	10	0	0	0.75	0.00	0.00
macaulay2:mgb	10	0	0	0.74	0.00	0.00
macaulay2:booleangb	0	10	0	—	—	—
m4gb	10	0	0	0.10	9.67	0.33
groebner.jl	10	0	0	8.09	392	2.13
openf4	3	0	7	0.17	0.00	0.00
sage:buchberger	10	0	0	178	117	1.01
sage:buchberger2	10	0	0	1.17	115	3.15
sage:d_basis	10	0	0	1.33	117	2.70

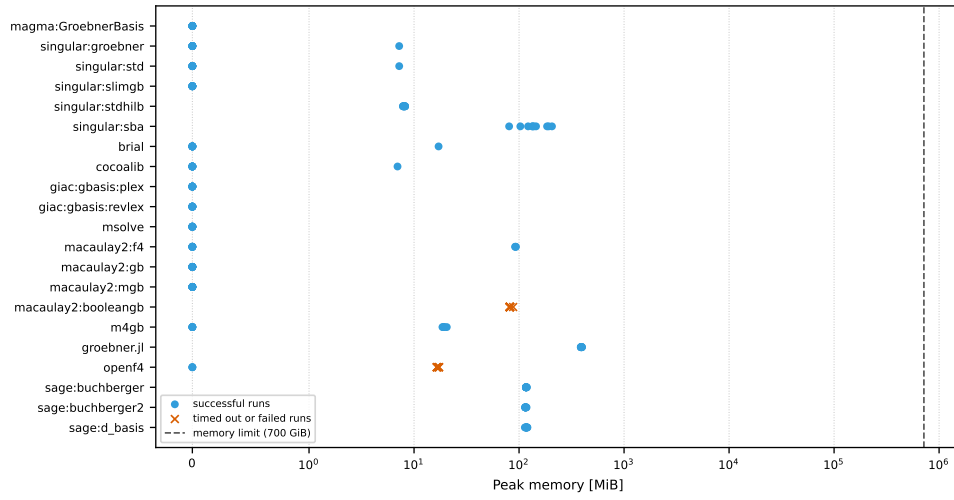
Time, memory, and CPU averages are computed over successful runs only.



■ **Figure 6.9** Runtime of the Pilot experiment with Boolean polynomials. Each point represents one key.

6.2.2 Easy Experiment With Boolean Polynomials

This experiment uses $\text{SR}(2, 2, 2, 4)$ with 8 PT-CT pairs and with Boolean polynomials. The results are shown in Table 6.6 and in Figures 6.11 and 6.12.



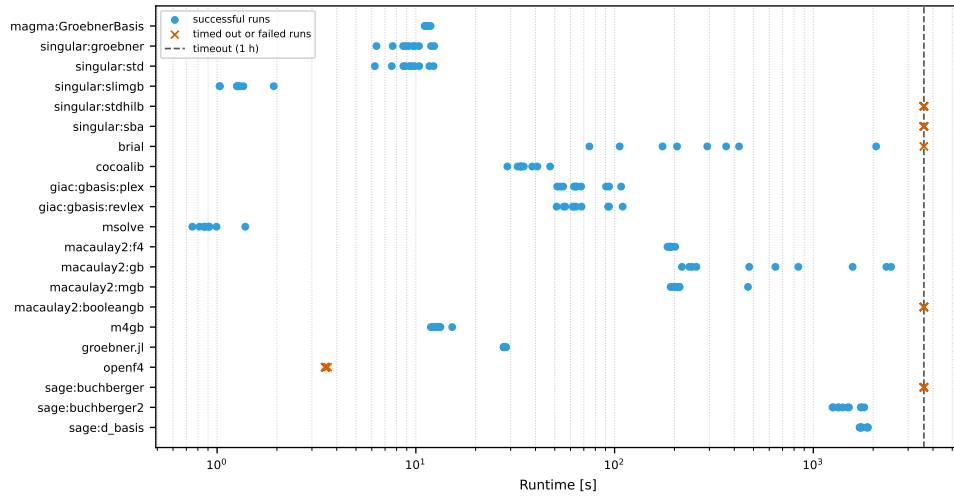
■ **Figure 6.10** Peak memory consumption of the Pilot experiment with Boolean polynomials. Each point represents one key.

■ **Table 6.6** Summary of the Easy experiment with Boolean polynomials

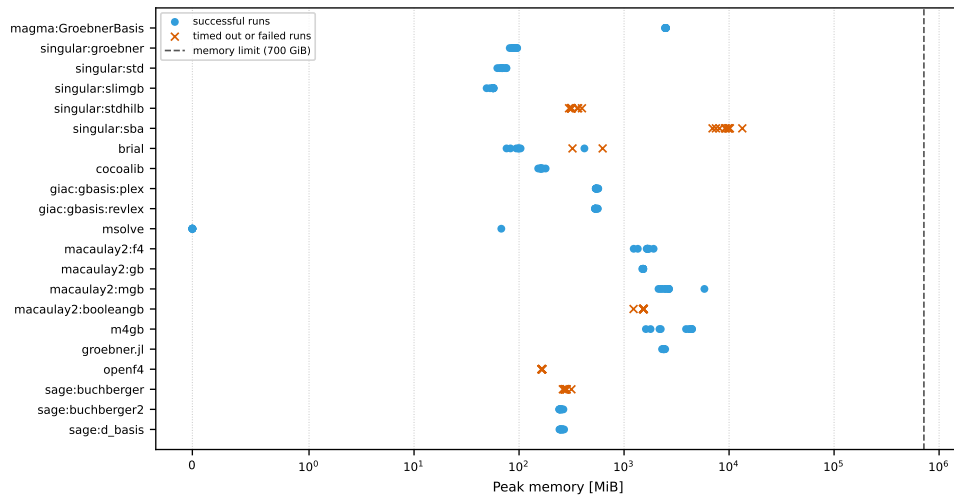
Implementation	OK	TO	Fail	Time [s]	Memory [MiB]	CPU
magma:GroebnerBasis	10	0	0	11.3	2487	0.98
singular:groebner	10	0	0	9.51	89.3	0.93
singular:std	10	0	0	9.43	69.5	0.94
singular:slimgb	10	0	0	1.30	55.5	0.77
singular:stdhilb	0	10	0	—	—	—
singular:sba	0	10	0	—	—	—
brial	8	2	0	464	134	1.00
cocoalib	10	0	0	35.8	163	0.98
giac:gbasis:plex	10	0	0	71.1	547	1.01
giac:gbasis:revlex	10	0	0	71.6	540	1.01
msolve	10	0	0	0.93	6.78	1.35
macaulay2:f4	10	0	0	191	1624	3.40
macaulay2:gb	10	0	0	929	1515	2.09
macaulay2:mgb	10	0	0	231	2808	3.02
macaulay2:booleangb	0	10	0	—	—	—
m4gb	10	0	0	13.0	3340	1.11
groebner.jl	10	0	0	27.9	2385	3.29
openf4	0	0	10	—	—	—
sage:buchberger	0	10	0	—	—	—
sage:buchberger2	10	0	0	1487	249	1.00
sage:d_basis	10	0	0	1788	253	1.00

Time, memory, and CPU averages are computed over successful runs only.

The overall performance mostly improved compared to the corresponding experiment without the Boolean polynomials. The implementations that



■ **Figure 6.11** Runtime of the Easy experiment with Boolean polynomials. Each point represents one key.



■ **Figure 6.12** Peak memory consumption of the Easy experiment with Boolean polynomials. Each point represents one key.

did not find any solution previously but found all solutions in this experiment are `singular:groebner`, `singular:std`, `cocoalib`, `macaulay2:gb`, `sage:buchberger2`, and `sage:d_basis`. Two implementations that regressed were again `macaulay2:booleangb` and `openf4`, which timed out and failed on all instances, respectively. Most of the remaining implementations improved their performance, either by recovering more solutions or by reducing runtime or memory consumption. The fastest implementation in this experiment was

`msolve`, with an average runtime of 0.93 seconds. Interestingly, the CPU usage of `msolve` dropped from 32.9 to 1.35. This is most likely because it found the solutions, on average, more than 120 times faster than before. Note that at these speeds, the memory and CPU metrics may not be accurate.

6.2.3 Medium Experiment With Boolean Polynomials

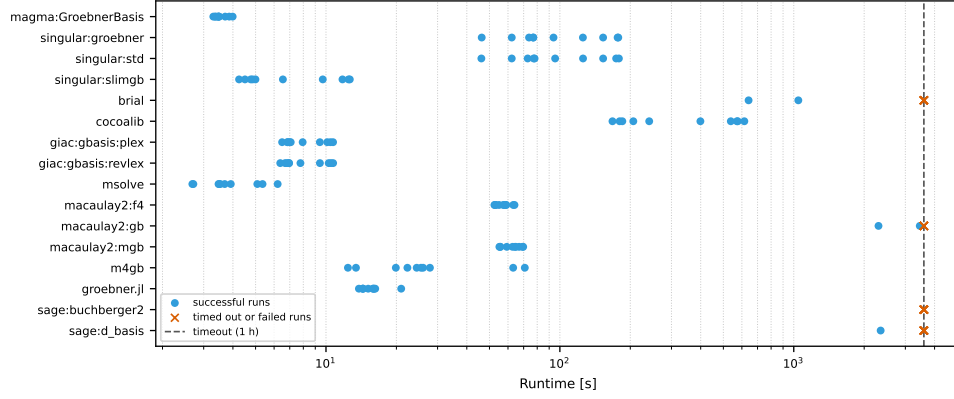
This experiment uses $\text{SR}(2, 2, 2, 4)$ with 2 PT-CT pairs and with Boolean polynomials. The results are shown in Table 6.7 and in Figures 6.13 and 6.14.

■ **Table 6.7** Summary of the Medium experiment with Boolean polynomials

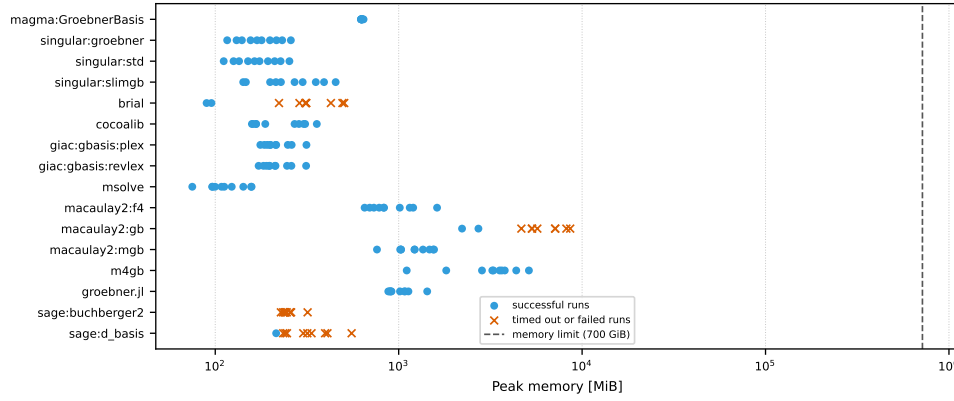
Implementation	OK	TO	Fail	Time [s]	Memory [MiB]	CPU
<code>magma:GroebnerBasis</code>	10	0	0	3.55	629	0.92
<code>singular:groebner</code>	10	0	0	106	179	0.99
<code>singular:std</code>	10	0	0	106	175	0.99
<code>singular:slimgb</code>	10	0	0	7.65	270	0.89
<code>brial</code>	2	8	0	844	92.5	1.00
<code>cocoalib</code>	10	0	0	368	237	1.00
<code>giac:gbasis:plex</code>	10	0	0	8.29	220	1.61
<code>giac:gbasis:revlex</code>	10	0	0	8.25	218	1.61
<code>msolve</code>	10	0	0	4.02	117	33.5
<code>macaulay2:f4</code>	10	0	0	57.4	950	4.20
<code>macaulay2:gb</code>	2	8	0	2880	2468	1.19
<code>macaulay2:mgb</code>	10	0	0	62.3	1221	2.75
<code>m4gb</code>	10	0	0	30.6	3278	1.27
<code>groebner.jl</code>	10	0	0	15.7	1022	3.83
<code>sage:buchberger2</code>	0	10	0	—	—	—
<code>sage:d_basis</code>	1	9	0	2352	215	1.00

Time, memory, and CPU averages are computed over successful runs only.

This experiment yielded substantially better results than the version without the extra polynomials. In this experiment, all solutions were found by `singular:groebner`, `singular:std`, `singular:slimgb`, `cocoalib`, both Giac implementations, `msolve`, `macaulay2:f4`, `macaulay2:mgb`, and `groebner.jl`. `macaulay2:gb` managed to recover two solutions and timed out on the remaining eight instances. Curiously, `sage:d_basis` was successful for one of the inputs, despite the fact that this is only a “toy” implementation. Previously, it timed out on all experiments except the Pilot experiment. `magma:GroebnerBasis`, `brial`, and `m4gb` saw almost no difference compared to the experiment without the polynomials, both in terms of runtime and memory consumption. The fastest implementation in this experiment was `magma:GroebnerBasis`, with an average runtime of 3.55 seconds. The only implementation that timed out on every instance was `sage:buchberger2`.



■ **Figure 6.13** Runtime of the Medium experiment with Boolean polynomials. Each point represents one key.



■ **Figure 6.14** Peak memory consumption of the Medium experiment with Boolean polynomials. Each point represents one key.

The three implementations that found only some solutions took the most time to do so, with averages of 2880, 2352, and 844 seconds. They are followed by `cocoalib` with 368 seconds on average. All the remaining implementations took around 100 seconds or less.

The memory consumption did not exceed 5 GiB for any of the implementations. The average CPU usage reached 33.5 cores for `msolve`, while the other implementations used at most 4.2 cores.

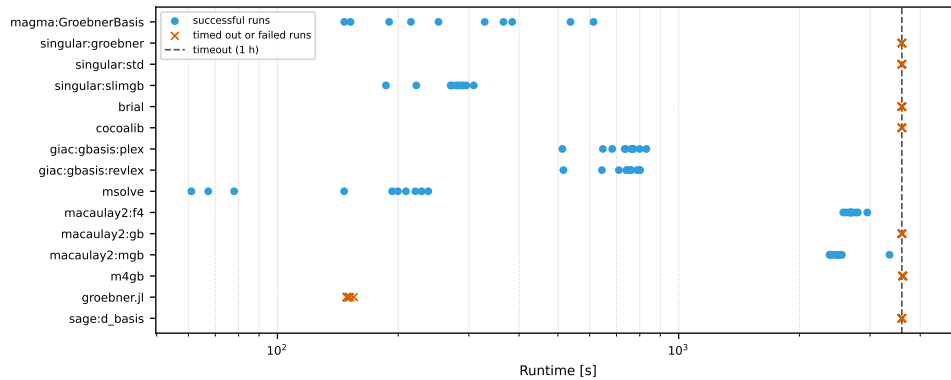
6.2.4 Hard Experiment With Boolean Polynomials

This last experiment uses $\text{SR}(2, 2, 4, 4)$ with 8 PT-CT pairs and with Boolean polynomials. The results are shown in Table 6.8 and in Figures 6.15 and 6.16.

■ **Table 6.8** Summary of the Hard experiment with Boolean polynomials

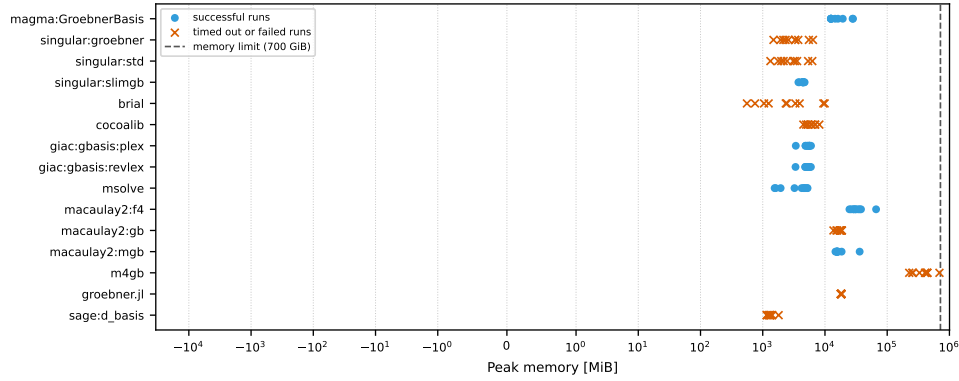
Implementation	OK	TO	Fail	Time [s]	Memory [MiB]	CPU
<code>magma:GroebnerBasis</code>	10	0	0	319	16853	1.00
<code>singular:groebner</code>	0	10	0	—	—	—
<code>singular:std</code>	0	10	0	—	—	—
<code>singular:slimgb</code>	10	0	0	269	4343	1.00
<code>brial</code>	0	10	0	—	—	—
<code>cocoalib</code>	0	10	0	—	—	—
<code>giac:gbasis:plex</code>	10	0	0	725	5206	1.49
<code>giac:gbasis:revlex</code>	10	0	0	726	5166	1.48
<code>msolve</code>	10	0	0	164	3693	13.4
<code>macaulay2:f4</code>	10	0	0	2713	34524	3.06
<code>macaulay2:gb</code>	0	10	0	—	—	—
<code>macaulay2:mgb</code>	10	0	0	2544	17931	2.69
<code>m4gb</code>	0	10	0	—	—	—
<code>groebner.jl</code>	0	0	10	—	—	—
<code>sage:d_basis</code>	0	10	0	—	—	—

Time, memory, and CPU averages are computed over successful runs only.



■ **Figure 6.15** Runtime of the Hard experiment with Boolean polynomials. Each point represents one key.

In the corresponding experiment without Boolean polynomials, the only successful implementation was Magma, which does not seem to be affected by the inclusion of the extra polynomials. However, adding those polynomials increased the number of open-source implementations capable of successful runs from zero to six: `singular:slimgb`, both Giac implementations, `msolve`, `macaulay2:f4`, and `macaulay2:mgb`. The longest runtimes were for the two Macaulay2 implementations, averaging around 2500 seconds. They are followed by the two Giac implementations, which averaged almost exactly 725 seconds. This suggests that there is no substantial observed performance difference between the two tested monomial orderings. The same pattern appears



■ **Figure 6.16** Peak memory consumption of the Hard experiment with Boolean polynomials. Each point represents one key.

in the previous experiments. `singular:slimgb` and `msolve` took 269 and 164 seconds, respectively, both beating the 319 seconds of `magma:GroebnerBasis`. The fastest implementation in this experiment was `msolve`.

The memory usage reached about 35 GiB for `macaulay2:f4` and around 17 GiB for `magma:GroebnerBasis` and `macaulay2:mgb`. The remaining implementations used around 5 GiB or less. The only notable CPU value is again for `msolve`, which decreased by more than half relative to the Medium experiment, to 13.4 cores.

6.3 Summary

We ran two sets of experiments, with and without the additional Boolean polynomials. Each set consisted of four experiments, each harder than the previous. Every experiment consisted of ten benchmark runs, each using a different key. The runtime varied greatly across the implementations, as did memory consumption and average number of CPU cores used.

Almost all implementations benefited from having the Boolean polynomials. The main regressions were `openf4` and `macaulay2:booleangb`. `openf4` failed with a nonzero return code on seven instances from the Pilot experiment and on all ten from the Easy experiment. `macaulay2:booleangb` timed out on all instances in the Boolean-polynomial experiments in which it was included. `magma:GroebnerBasis`, `brial`, and `m4gb` performed almost the same, whether they were provided with the additional polynomials or not.

The implementations focused on Gröbner basis computation are data-sensitive. The choice of the small-scale AES variant and the number of PT-CT pairs have a substantial impact on the runtime of these algorithms. As observed in the Easy and Medium experiments with Boolean polynomials, many implementations performed better with more pairs, while others, such as

`magma:GroebnerBasis` or `macaulay2:mgb`, benefited from fewer pairs. Moreover, the runtime can differ even between input instances of the same input variant. An example of this is the behavior of `brial` in both Medium experiments. It found two solutions in under a quarter of an hour, while the remaining eight input instances exceeded the one-hour timeout.

Due to this inconsistent nature of the difficulty of the inputs, it is difficult to make a general and definitive statement about the performance of these implementations, as it is possible that for some other input variant the results could be different. Nevertheless, based on these experiments, multiple implementations were capable of Gröbner basis computation for even moderately difficult inputs.

In the experiments without the Boolean polynomials, the best results from the open-source implementations were observed in the Medium experiment. Here, `brial` recovered two keys, and `macaulay2:booleangb` and `m4gb` recovered all ten. Furthermore, `m4gb` did so more than 20 times faster than `macaulay2:booleangb`. All three were outperformed by `magma:GroebnerBasis`, which was 6 times faster than even `m4gb`. It was also the only implementation that managed to pass all instances in the Hard experiment, while the other three remaining implementations always timed out.

After adding the extra Boolean polynomials, twelve implementations solved all instances in the Medium experiment. Moving to the Hard experiment, seven implementations found all solutions, while the remaining implementations we tested either timed out or failed with a nonzero return code in the case of `groebner.jl`. Among these implementations, `macaulay2:f4` and `macaulay2:mgb` took on average about 45 minutes. Both Giac variants followed with around 700 seconds. The three fastest implementations were `magma:GroebnerBasis` at 319 seconds, `singular:slimgb` at 269 seconds, and `msolve` at 164 seconds.

Multiple implementations proved capable of finding Gröbner bases. We also showed that the inclusion of Boolean polynomials in most cases led to much better performance. Ultimately, based on these results, we identify `msolve` as the best-performing implementation in our experiments. Not only did it solve the hardest inputs fastest, but it also did so while using the least memory, at around 3700 MiB. It achieved these results through heavy use of multithreading.

Improving msolve

The goal of this chapter is to attempt to improve one of the implementations. We chose `msolve`, as it performed the best in our experiments, so making it even faster is a desirable goal. Since the initial commit in 2021, the project has been under active development, with the newest commit being 5 days old at the time of writing [58]. Due to this, the project is complex. Examining it as a whole is beyond the scope of this work. Instead, we focused only on the part of the code that gets executed in our setting and is the most computationally intensive. We then attempted to identify parts of the code that could be improved. We now present the combined optimization, which consists of two parts: caching metadata of leading monomials and using this cached metadata for an additional degree-based filter during reducer lookup. All changes were applied to the most recent commit `7e0ef99d`. At this commit, the project is in version 0.9.5, compared to the 0.9.4 we examined in the previous chapters. The implementation changes presented in this work are available in the *thesis* branch of the repository [110].

7.1 Caching Leading Monomial Metadata

The first improvement was applied in the symbolic preprocessing phase of the F4 algorithm, implemented in `src/neogb/symbol.c`. In F4, symbolic preprocessing constructs the set of monomials and reducer rows that will form the matrix for linear algebra. During this phase, the function `find_multiplied_reducer` is repeatedly called for monomials that occur in the symbolic matrix. Its purpose is to find a polynomial in the current basis whose leading monomial divides the target monomial. Such a polynomial is called a reducer, because it can be multiplied by the monomial obtained as the quotient of the target monomial and the reducer’s leading monomial. The resulting multiple is then inserted as a row into the F4 matrix, where it can reduce or cancel the target monomial during row reduction. For example, if

the target monomial is x^3y^2 and a basis polynomial has leading monomial xy , then the quotient monomial is x^2y , since $x^2y \cdot xy = x^3y^2$.

We first describe the data structures used by the relevant code. The variable **bs** is the current basis, represented by the structure **bs_t**. Its field **bs->hm** stores basis polynomials as arrays of hashed monomials. A hashed monomial is not stored directly as a string or as an exponent vector inside a polynomial row; instead, it is represented by an integer index into a monomial hash table. Given such an index, the actual exponent vector of the monomial can be obtained from the hash table. In this code, the type **hm_t** is used for such hashed monomial indices. The first entries of each polynomial array contain metadata, while the actual monomials start at index **OFFSET**. Therefore, **bs->hm[j][OFFSET]** is the hashed leading monomial of the *j*-th basis polynomial.

The basis can contain polynomials that are no longer needed for generating the current leading ideal. Such elements are marked as redundant. The code therefore maintains a compact list of non-redundant leading monomials, meaning leading monomials of basis elements that have not been marked redundant and are still used as possible reducers. The field **bs->lm** stores short divisor masks of these leading monomials, and **bs->lmps** stores the corresponding positions of the basis polynomials. A short divisor mask is a compact bit mask used as a cheap necessary test for monomial divisibility. If the mask of a candidate divisor contains a bit that is not present in the mask of the target monomial, then the candidate cannot divide the target. Passing this mask test does not prove divisibility; it only means that the full exponent-vector comparison is still necessary. Thus, **bs->lmps[i]** maps the index *i* in the compact list of non-redundant leading monomials to the position of the corresponding polynomial in **bs->hm**. Consequently, **bs->hm[bs->lmps[i]][OFFSET]** is the hashed monomial representing the *i*-th non-redundant leading monomial. The field **bs->lm1** stores the length of this compact list, i.e., the number of non-redundant leading monomials.

The hash table of the basis is stored in **bs->ht**; in Code listing 7.1, it is assigned to the local variable **bht**. The array **bht->ev** contains exponent vectors indexed by hashed monomials, and **bht->hd** contains metadata about these monomials. This metadata includes, among other values, the total degree **deg**, the short divisor mask **sdm**, and the hash value **val**. The current symbolic hash table is stored in **md->ht**, assigned locally to **sht**. This table contains monomials that occur in the current symbolic preprocessing pass.

The basis hash table **bht** and the symbolic hash table **sht** serve different purposes. The basis hash table belongs to the current basis and stores monomials occurring in basis polynomials. In contrast, the symbolic hash table is temporary and belongs to the current symbolic preprocessing pass; it stores monomials that occur in the F4 matrix being constructed. Therefore, when **find_multiplied_reducer** compares a target monomial with a candidate leading monomial, the target monomial is read from **sht**, while the candidate leading monomial comes from the basis and is read from **bht**.

The original implementation repeatedly followed the chain `bs->hm`, `bs->lmps`, and `bht->ev` inside reducer lookup. This means that for each candidate leading monomial and for each target monomial, the code re-computed where the exponent vector of the candidate leading monomial is stored. The optimization caches the exponent vectors and total degrees of all current non-redundant leading monomials once at the beginning of `symbolic_preprocessing`. This cache construction is shown in Code listing 7.1.

■ **Code listing 7.1** Caching leading monomial exponent vectors and degrees

```

1 ht_t *bht = bs->ht;
2
3 const len_t lml = bs->lml;
4 const exp_t **lead_evs = (const exp_t **)malloc(
5     (uint64_t)lml * sizeof(exp_t *));
6 deg_t *lead_degs = (deg_t *)malloc((uint64_t)lml * sizeof(deg_t));
7
8 for (i = 0; i < lml; ++i) {
9     const hm_t lead_hash = bs->hm[bs->lmps[i]][OFFSET];
10    lead_evs[i] = bht->ev[lead_hash];
11    lead_degs[i] = bht->hd[lead_hash].deg;
12 }
```

In Code listing 7.1, `lml` is the number of non-redundant leading monomials in the current basis. The array `lead_evs` stores pointers to the exponent vectors of these leading monomials, and `lead_degs` stores their total degrees. The variable `lead_hash` is a hashed monomial, i.e., an integer index into the basis monomial hash table. The expression `bht->ev[lead_hash]` obtains the exponent vector of this monomial, and `bht->hd[lead_hash].deg` obtains its total degree. The arrays are valid during one symbolic preprocessing pass, because the list of non-redundant leading monomials is fixed during that pass.

The cached arrays are then passed to `find_multiplied_reducer`, as shown in Code listing 7.2.

■ **Code listing 7.2** Passing cached leading monomial metadata to reducer lookup

```

1 find_multiplied_reducer(bs, i, &nrr, mat->rr, sht, md,
2     lead_evs, lead_degs);
```

In Code listing 7.2, the variable `i` is a hash index of the target monomial in the current symbolic hash table `sht`. The variable `nrr` is the number of reducer rows found so far, and `mat->rr` is the array of reducer rows of the

F4 matrix. Passing `lead_evs` and `lead_degs` gives the reducer lookup direct access to cached metadata of the current leading monomials.

The signature of `find_multiplied_reducer` was extended to accept the cached arrays, as shown in Code listing 7.3.

■ **Code listing 7.3** Extended reducer lookup signature

```

1 static inline void find_multiplied_reducer(
2     bs_t *bs,
3     const hm_t m,
4     len_t *nr,
5     hm_t **rows,
6     ht_t *sht,
7     const md_t * const md,
8     const exp_t * const * const lead_evs,
9     const deg_t * const lead_degs
10 )

```

In Code listing 7.3, `m` is the target monomial for which a reducer is being searched. The variable `rows` points to the reducer rows of the current matrix, and `nr` points to the number of rows already inserted. The newly added arguments `lead_evs` and `lead_degs` provide cached leading monomial metadata.

Inside the function, the cached exponent vector is used directly, as shown in Code listing 7.4.

■ **Code listing 7.4** Using the cached exponent vector of a leading monomial

```

1 const hm_t *b = bs->hm[lmps[i]];
2 const hm_t lead = b[OFFSET];
3 const exp_t * const f = lead_evs[i];

```

Here, `lmps` is a local pointer to `bs->lmps`. Therefore, `b` is the basis polynomial corresponding to the `i`-th non-redundant leading monomial. The variable `lead` is the hashed leading monomial of this polynomial. The variable `f` is the exponent vector of that leading monomial. In the original implementation, `f` was obtained by looking up `bht->ev[b[OFFSET]]` each time. The new version avoids this repeated lookup by using `lead_evs[i]`.

This change does not alter the mathematical behavior of the algorithm. It only avoids repeated pointer chasing through the basis and hash-table data structures. The exact divisibility test on exponent vectors remains unchanged.

7.2 Using Cached Degrees for Reducer Filtering

The second part of the optimization adds an additional necessary condition for monomial divisibility. If a monomial α divides a monomial β , then the total degree of α cannot be larger than the total degree of β . Therefore, when searching for a reducer whose leading monomial divides a target monomial, any candidate with a larger total degree can be skipped before performing the full exponent-vector comparison.

The original code already used short divisor masks as a cheap pre-filter. The local variable `lms` points to `bs->lm`, which stores short divisor masks of the non-redundant leading monomials of the basis. These masks are used only as a one-sided filter: they can rule out impossible divisors, but they cannot confirm divisibility. The metadata for the target monomial is stored in `hdm`, which is obtained from the symbolic hash table `sht`. The added degree-based filter is shown in Code listing 7.5.

■ **Code listing 7.5** Degree-based filtering of reducer candidates

```

1  const hd_t hdm = sht->hd[m];
2  const len_t lml = bs->lml;
3  const sdm_t ns = ~hdm.sdm;
4  const deg_t target_deg = hdm.deg;
5
6  const sdm_t * const lms = bs->lm;
7
8  while (i < lml && ((lms[i] & ns) || lead_degs[i] > target_deg)) {
9      i++;
10 }
```

In Code listing 7.5, `hdm.sdm` is the short divisor mask of the target monomial `m`, and `target_deg` is its total degree. The expression `lms[i] & ns` is nonzero if the divisor mask of the candidate leading monomial contains a bit not present in the target monomial. In that case, the candidate cannot divide the target monomial and is skipped. The second condition, `lead_degs[i] > target_deg`, is the new degree-based filter. If the leading monomial of a candidate reducer has total degree larger than that of the target monomial, it cannot divide it. Since `lead_degs` was cached at the beginning of symbolic preprocessing, this additional check is cheap in the combined version.

Only candidates that pass these cheap filters are checked using the full component-wise exponent-vector comparison. This unchanged exact check is shown in Code listing 7.6.

In Code listing 7.6, `e` is the exponent vector of the target monomial `m`, and `f` is the exponent vector of the candidate leading monomial. The variable `evl` is the length of exponent vectors. The condition `e[k] < f[k]` means that the

■ **Code listing 7.6** Exact component-wise monomial divisibility check

```

1 for (k = 0; k < evl; ++k) {
2     if (e[k] < f[k]) {
3         i++;
4         goto start;
5     }
6     etmp[k] = (exp_t)(e[k] - f[k]);
7 }
```

candidate leading monomial has a larger exponent in at least one variable, so it cannot divide the target monomial. If this happens, the function moves to the next candidate. If all components pass, the difference $\mathbf{e} - \mathbf{f}$ is stored in **etmp**. This vector represents the monomial multiplier by which the reducer must be multiplied so that its leading monomial matches the target monomial.

The degree-based filter is conservative: it only rejects candidates that cannot possibly divide the target monomial. The original exact exponent-vector divisibility test is still executed for all remaining candidates. Consequently, the set of valid reducers is unchanged.

7.3 Results

To test the performance of these changes, we used the same inputs as in the Hard experiment with Boolean polynomials. This choice is based on the results from Chapter 6: the practical use case for this implementation involves inputs with Boolean polynomials, and among these inputs, only the Hard experiment took longer than several seconds to complete.

To distinguish between the original **msolve** in version 0.9.4 and version 0.9.5, which we use as the baseline, version 0.9.5 is named **msolve-new** in the following experiment. **msolve-new-improved** is built from the same version, but includes the two improvements described above. The results are summarized in Tables 7.1, 7.2, and 7.3, and visualized in Figures 7.1, 7.2, and 7.3.

■ **Table 7.1** **msolve** runtime improvement comparison on the Hard experiment with Boolean polynomials

Implementation	0	1	2	3	4	5	6	7	8	9	Average
msolve-new	134	202	211	47.3	42.5	45.8	183	192	224	232	151
msolve-new-improved	95.8	142	150	40.8	36.9	39.1	129	136	160	165	110
Improved/original	0.72	0.70	0.71	0.86	0.87	0.85	0.71	0.71	0.71	0.71	0.76

Values are reported in s; the ratio row is **msolve-new-improved** divided by **msolve-new**.

■ **Table 7.2** msolve peak memory consumption improvement comparison on the Hard experiment with Boolean polynomials

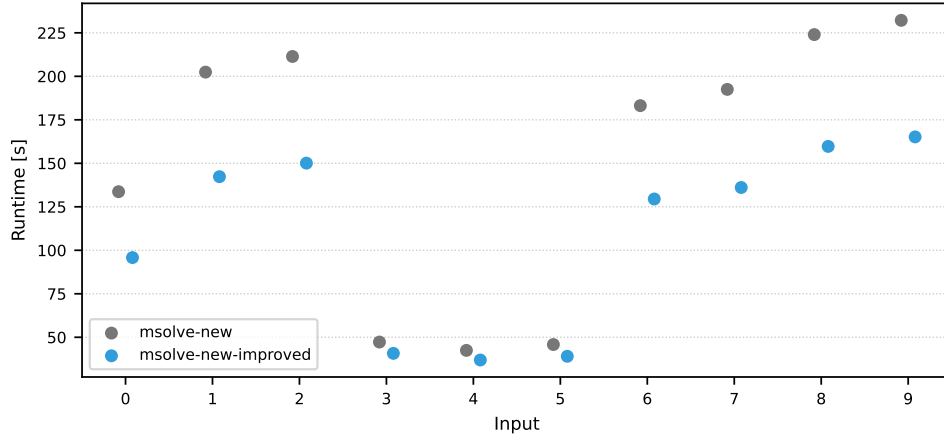
Implementation	0	1	2	3	4	5	6	7	8	9	Average
msolve-new	3237	4641	4817	1966	1555	1610	4217	4510	5149	5249	3695
msolve-new-improved	3236	4638	4820	1942	1556	1611	4216	4510	5149	5250	3693
Improved/original	1.00	1.00	1.00	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Values are reported in MiB; the ratio row is `msolve-new-improved` divided by `msolve-new`.

■ **Table 7.3** msolve CPU usage improvement comparison on the Hard experiment with Boolean polynomials

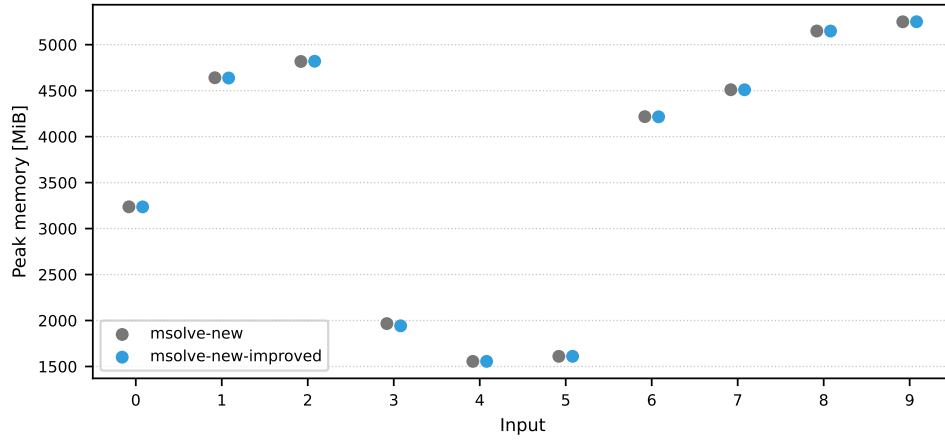
Implementation	0	1	2	3	4	5	6	7	8	9	Average
msolve-new	5.07	4.10	3.65	14.9	14.3	13.8	3.96	3.82	3.46	3.38	7.05
msolve-new-improved	6.63	4.88	4.66	17.2	16.4	15.9	5.24	5.02	5.02	4.95	8.60
Improved/original	1.31	1.19	1.28	1.15	1.15	1.15	1.33	1.31	1.45	1.47	1.28

Values are reported in cores; the ratio row is `msolve-new-improved` divided by `msolve-new`.

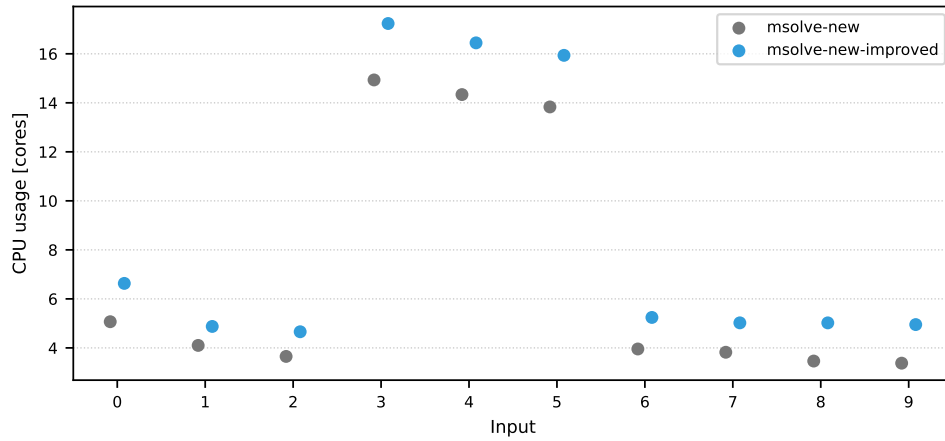


■ **Figure 7.1** msolve runtime improvement comparison on the Hard experiment with Boolean polynomials

The most important metric is the runtime, which on average decreased to 76% of the baseline. Considering that msolve was already the best-performing implementation, this improvement is substantial. The modification improved the implementation not only on average, but also on each individual input instance. The smallest observed decrease was to 87% of the original runtime, while the largest took only 70% of the original runtime. This is roughly equivalent to a 43% speed increase ($0.70^{-1} = 1.429$). Taking into account the simplicity of the implemented changes, this is a notable result. The improved variant also made greater use of multithreading. The memory consumption remained essentially unchanged.



■ **Figure 7.2** msolve peak memory consumption improvement comparison on the Hard experiment with Boolean polynomials



■ **Figure 7.3** msolve CPU usage improvement comparison on the Hard experiment with Boolean polynomials

Interestingly, it is not clear how much each of the two changes contributed to the improvement. We tested the first input instance in all four combinations: without modifications, only with caching, only with reducer filtering without the accompanying caching improvement, and with both improvements. We tested it on one additional machine equipped with the *AMD Ryzen 7 5800X3D* processor with 8 physical cores, 16 logical cores, and 128 GiB of RAM. The results are shown in Table 7.4.

Using only reducer filtering made the implementation slow down in both cases. Using only caching on the Ryzen system led to almost the same performance gain as using both modifications; the difference may be due to mea-

■ **Table 7.4** msolve modification comparison on two systems

Modification	Xeon time [s]	Ryzen time [s]	Xeon relative	Ryzen relative
No modifications	132.27	102.48	–	–
Only caching	123.97	64.91	0.94	0.63
Only reducer filtering	135.75	106.41	1.03	1.04
Both modifications	96.97	65.19	0.73	0.64

Relative values are computed with respect to the unmodified baseline on the same system.

surement noise. One might therefore conclude that all the performance gain was achieved solely by caching and that reducer filtering did not contribute anything, perhaps even making performance worse.

However, this is not what we observed on the Xeon system. There, while using only the caching modification improved the results, the improvement was only slight. The main improvement appeared only when using both modifications. We recompiled all the implementations and double-checked the results, and they are indeed correct. One possible explanation is that the Xeon system has 24.75 MB of L3 cache per CPU [111], so 49.5 MB in total, while the Ryzen system has 96 MB of L3 cache [112]. This could explain why the Ryzen system benefited more from caching.

We did not examine this further, nor did we run further tests, as determining how much each modification contributed is not ultimately relevant here: the combined variant produced the main improvement on the Xeon system and almost the same result as caching alone on the Ryzen system. Interestingly, for this input instance on the Ryzen system, the runtime with both modifications decreased to only 64% of the original runtime, which is a larger improvement than any observed on the Xeon system.

The modifications are available in the public repository cited at the beginning of this chapter. We intend to share these changes with the authors of msolve.

Conclusion

This work aimed to identify existing open-source implementations capable of Gröbner basis computation and evaluate them using inputs derived from small-scale AES instances. We provided a critical discussion of the performance of the different implementations. Finally, we proposed improvements to the msolve implementation, which reduced its runtime in the experiments.

In Chapter 2, we introduced the mathematical background required for understanding Gröbner bases. We also explained the F4 algorithm. Chapter 3 presented the small-scale AES framework, its parameters, the round operations of the cipher, and the basic idea behind deriving polynomial systems from known plaintext-ciphertext pairs for recovering the secret key.

In Chapter 4, we listed 20 open-source implementations and the proprietary reference system Magma. Where the information was available, we described, for each implementation, the algorithm it uses, the available monomial orderings, why we selected a particular ordering, how to run it, and provided example input and output files.

In Chapter 5, we described the benchmarking framework we developed for the experiments in this work. It consists of three parts: input generation, experimental evaluation of the implementations, and collection of performance metrics. The framework is highly configurable and modular, allowing fine control over system resources and parallel execution of the implementations. It also records metrics such as runtime, peak memory consumption, and average number of CPU cores used. Although it was designed primarily for the evaluation of the Gröbner basis implementations, its modularity makes it easy to use for other benchmarking purposes with little to no modification.

In Chapter 6, we used the benchmarking framework to evaluate the implementations. The evaluation used four different input variants determined by the small-scale AES variant and the number of plaintext-ciphertext pairs used during the generation of the polynomial system. We also performed all experiments twice, once with and once without the added Boolean polynomials. The experiments show that adding these polynomials improves performance in

most cases. In the hardest experiment with Boolean polynomials, two open-source implementations outperformed Magma in average runtime: *msolve* and the *slimgb* algorithm from Singular. Compared to Magma, the average runtime was 16% lower for *slimgb* and 49% lower for *msolve*.

In Chapter 7, we took the best-performing implementation, *msolve*, and proposed two improvements. The first caches data about the leading monomials, and the second adds a degree-based filter during reducer lookup. Benchmarking these modifications on the Hard experiment with Boolean polynomials resulted in lower runtime across all input instances. Compared to the unoptimized *msolve* implementation, the mean per-instance runtime decrease was 24%, and the largest observed decrease was 30%. We observed an even larger improvement on a different hardware configuration, but this observation was outside the scope of the work and was not analyzed further.

We cannot make a general statement about the performance of the implementations, because runtimes are strongly affected by the inputs; for other inputs, the results could differ. However, based on our results, we conclude that certain open-source implementations are competitive alternatives to proprietary systems such as Magma, especially *msolve* and Singular. Future work could include running the same experiments on even harder inputs or analyzing the *msolve* improvements in more detail, especially the performance inconsistencies observed across the two tested hardware configurations when the improvements were applied individually.

Bibliography

1. NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. *Advanced Encryption Standard (AES)*. 2023-05. Federal Information Processing Standards Publication, NIST FIPS 197-upd1. National Institute of Standards and Technology. Available from DOI: 10.6028/NIST.FIPS.197-upd1.
2. COURTOIS, Nicolas; KLIMOV, Alexander; PATARIN, Jacques; SHAMIR, Adi. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In: *Advances in Cryptology — EUROCRYPT 2000*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 392–407. Lecture Notes in Computer Science.
3. BERUŠKOVÁ, Jana; JUREČEK, Martin; JUREČKOVÁ, Olha. Reducing Overdefined Systems of Polynomial Equations Derived from Small Scale Variants of the AES via Data Mining Methods. *Journal of Computer Virology and Hacking Techniques*. 2024, vol. 20, no. 4, pp. 885–900. ISSN 2263-8733. Available from DOI: 10.1007/s11416-024-00540-2.
4. CID, Carlos; MURPHY, Sean; ROBSHAW, Matthew J. B. Small Scale Variants of the AES. In: *Fast Software Encryption*. Berlin, Heidelberg: Springer, 2005, vol. 3557, pp. 145–162. ISBN 978-3-540-26541-2. Available from DOI: 10.1007/11502760_10.
5. COMPUTATIONAL ALGEBRA GROUP. *Magma Computational Algebra System* [online]. 2026. [visited on 2026-04-21]. Available from: <https://magma.maths.usyd.edu.au/magma/>.
6. *Multivariate Polynomial Systems Solving - msolve library* [online]. [visited on 2026-04-22]. Available from: <https://msolve.lip6.fr/>.
7. COX, David A.; LITTLE, John; O’SHEA, Donal. *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra*. Cham, Switzerland: Springer Nature Switzerland, 2025. ISBN 9783031918414. ISSN 2197-5604. Available from DOI: 10.1007/978-3-031-91841-4.

8. BIELIK, Marek; JUREČEK, Martin; JUREČKOVÁ, Olha; LÓRENCZ, Róbert. Yet Another Algebraic Cryptanalysis of Small Scale Variants of AES. In: *Proceedings of the 19th International Conference on Security and Cryptography – SECRYPT*. SCITEPRESS – Science and Technology Publications, 2022, pp. 415–427. ISBN 978-989-758-590-6. ISSN 2184-7711. Available from DOI: 10.5220/0011327900003283.
9. WIKIPEDIA CONTRIBUTORS. *Advanced Encryption Standard* [online]. [visited on 2026-05-01]. Available from: https://en.wikipedia.org/wiki/Advanced_Encryption_Standard.
10. BARD, Gregory V. *Algebraic Cryptanalysis*. New York, NY: Springer, 2009. ISBN 978-0-387-88757-9. Available from DOI: 10.1007/978-0-387-88757-9.
11. CID, Carlos; MURPHY, Sean; ROBSHAW, Matthew J. B. *Algebraic Aspects of the Advanced Encryption Standard*. New York, NY: Springer, 2006. ISBN 978-0-387-24363-4. Available from DOI: 10.1007/978-0-387-36842-9.
12. BIELIK, Marek. *YAAC of AES: experiments* [online]. 2020. [visited on 2026-04-26]. Available from: <https://gitlab.com/upbqdn/yaac/-/tree/main/experiments>.
13. FAUGÈRE, Jean-Charles. FGb: A Library for Computing Gröbner Bases. In: FUKUDA, Komei; HOEVEN, Joris van der; JOSWIG, Michael; TAKAYAMA, Nobuki (eds.). *Mathematical Software – ICMS 2010*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 84–87. ISBN 978-3-642-15582-6.
14. KREDEL, Heinz. *Java Algebra System (JAS)* [online]. 2023. [visited on 2026-04-28]. Available from: <https://krum.rz.uni-mannheim.de/jas/>.
15. SAGEMATH. *SageMath - Open-Source Mathematical Software System* [online]. [visited on 2026-04-21]. Available from: <https://www.sagemath.org/>.
16. THE SAGE DEVELOPMENT TEAM. *Sage Reference Manual: MPolynomialIdeal.groebner_basis* [online]. 2025. [visited on 2026-04-21]. Available from: https://doc.sagemath.org/html/en/reference/polynomial_rings/sage/rings/polynomial/multi_polynomial_ideal.html#sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal.groebner_basis.
17. THE SAGE DEVELOPMENT TEAM. *Sage Reference Manual: BooleanPolynomialIdeal.groebner_basis* [online]. 2025. [visited on 2026-04-21]. Available from: https://doc.sagemath.org/html/en/reference/polynomial_rings/sage/rings/polynomial/pbri/pbri.html#sage.rings.polynomial.pbri.pbri.BooleanPolynomialIdeal.groebner_basis.

18. FREE SOFTWARE FOUNDATION. *GNU General Public License, version 3* [online]. 2007. [visited on 2026-04-21]. Available from: <https://www.gnu.org/licenses/gpl-3.0.html>.
19. DECKER, Wolfram; GREUEL, Gert-Martin; PFISTER, Gerhard; SCHÖNEMANN, Hans. *Singular* [online]. [visited on 2026-04-21]. Available from: <https://www.singular.uni-kl.de/>.
20. *Singular Manual: libSingular* [online]. 2021. [visited on 2026-04-21]. Available from: https://www.singular.uni-kl.de/Manual/4-2-1/sing_3000.htm.
21. *Singular Manual: The SINGULAR language* [online]. 2025. [visited on 2026-04-21]. Available from: https://www.singular.uni-kl.de/hannes/sing_34.htm#SEC56.
22. *Singular Manual: Command line options* [online]. 2025. [visited on 2026-04-21]. Available from: https://www.singular.uni-kl.de/hannes/sing_19.htm#SEC26.
23. *Singular Manual: Computing Groebner and Standard Bases* [online]. 2025. [visited on 2026-04-21]. Available from: https://www.singular.uni-kl.de/hannes/sing_942.htm.
24. *Singular Manual: stdhilb* [online]. 2022. [visited on 2026-04-21]. Available from: https://www.singular.uni-kl.de/Manual/4-3-1/sing_359.htm.
25. *Singular Manual: Term orderings* [online]. 2016. [visited on 2026-04-21]. Available from: https://www.singular.uni-kl.de/Manual/4-0-3/sing_31.htm.
26. *Singular Manual: sba* [online]. 2025. [visited on 2026-04-21]. Available from: https://www.singular.uni-kl.de/hannes/sing_353.htm#SEC393.
27. *Singular Manual: std* [online]. 2025. [visited on 2026-04-21]. Available from: https://www.singular.uni-kl.de/hannes/sing_364.htm#SEC404.
28. *Singular Manual: slimgb* [online]. 2025. [visited on 2026-04-21]. Available from: https://www.singular.uni-kl.de/hannes/sing_358.htm#SEC398.
29. *Singular Manual: groebner* [online]. 2025. [visited on 2026-04-21]. Available from: https://www.singular.uni-kl.de/hannes/sing_266.htm#SEC306.
30. NUMPY TEAM. *NumPy* [online]. 2026. [visited on 2026-04-21]. Available from: <https://numpy.org/>.
31. SCIPY COMMUNITY. *SciPy* [online]. 2026. [visited on 2026-04-21]. Available from: <https://scipy.org/>.

32. MATPLOTLIB DEVELOPMENT TEAM. *Matplotlib: Visualization with Python* [online]. 2026. [visited on 2026-04-21]. Available from: <https://matplotlib.org/>.
33. THE SAGE DEVELOPMENT TEAM. *Sage Tutorial: The Interactive Shell* [online]. 2025. [visited on 2026-04-21]. Available from: https://doc.sagemath.org/html/en/tutorial/interactive_shell.html.
34. THE SAGE DEVELOPMENT TEAM. *Sage Reference Manual: Constructors for polynomial rings* [online]. 2025. [visited on 2026-04-21]. Available from: https://doc.sagemath.org/html/en/reference/polynomial_rings/sage/rings/polynomial/polynomial_ring_constructor.html#sage.rings.polynomial.polynomial_ring_constructor.PolynomialRing.
35. ALBRECHT, Martin; HAMPTON, Marshall; TEAM, The Sage Development. *Sage Reference Manual: Educational Versions of Groebner Basis Algorithms* [online]. 2025. [visited on 2026-04-21]. Available from: https://doc.sagemath.org/html/en/reference/polynomial_rings/sage/rings/polynomial/toy_buchberger.html#sage.rings.polynomial.toy_buchberger.
36. THE SAGE DEVELOPMENT TEAM. *Sage GitHub repository: multi_polynomial_ideal.py* [online]. 2026. [visited on 2026-04-21]. Available from: https://github.com/sagemath/sage/blob/develop/src/sage/rings/polynomial/multi_polynomial_ideal.py.
37. BECKER, Thomas; WEISPFENNING, Volker. *Gröbner Bases*. Springer New York, 1993. ISBN 9781461209133. ISSN 0072-5285. Available from DOI: 10.1007/978-1-4612-0913-3.
38. ALBRECHT, Martin. *Sage Reference Manual: Educational Version of the d-Groebner Basis Algorithm over PIDs* [online]. 2025. [visited on 2026-04-21]. Available from: https://doc.sagemath.org/html/en/reference/polynomial_rings/sage/rings/polynomial/toy_d_basis.html.
39. DREYER, Alexander; BRICKENSTEIN, Michael. *PolyBoRi - Polynomials over Boolean Rings: Overview* [online]. [visited on 2026-04-21]. Available from: <https://polybori.sourceforge.net/index.html>.
40. THE SAGE DEVELOPMENT TEAM. *Sage Package Reference: brial* [online]. 2025. [visited on 2026-04-21]. Available from: <https://doc.sagemath.org/html/en/reference/spkg/brial.html>.
41. DREYER, Alexander. *PolyBoRi SourceForge files: polybori 0.8.3* [online]. 2013. [visited on 2026-04-21]. Available from: <https://sourceforge.net/projects/polybori/files/polybori/0.8.3/>.
42. BRIAL. *BRiAl GitHub repository* [online]. 2025. [visited on 2026-04-21]. Available from: <https://github.com/BRiAl/BRiAl>.

43. DREYER, Alexander. *BRiAl GitHub repository: pbori_order.h* [online]. 2006. [visited on 2026-04-21]. Available from: https://github.com/BRiAl/BRiAl/blob/master/libbrial/include/polybori/orderings/pbori_order.h.
44. DREYER, Alexander. *BRiAl GitHub repository: BoolePolyRing.h* [online]. 2006. [visited on 2026-04-21]. Available from: <https://github.com/BRiAl/BRiAl/blob/master/libbrial/include/polybori/BoolePolyRing.h>.
45. *CoCoA System* [online]. [visited on 2026-04-22]. Available from: <https://sites.google.com/view/cocoa-cocoalib>.
46. COCOA-OFFICIAL. *CoCoALib* [online]. 2026. [visited on 2026-04-22]. Available from: <https://github.com/cocoa-official/CoCoALib>.
47. *CoCoA Manual: CoCoA Tutorials and Programming Language* [online]. [visited on 2026-04-22]. Available from: <https://cocoa.altervista.org/html/toc.html#2>.
48. ABBOTT, John; BIGATTI, Anna. Groebner Bases for Everyone with CoCoA-5 and CoCoALib. 2016. Available from DOI: 10.48550/arXiv.1611.07306.
49. *CoCoA Manual: Term Orderings* [online]. [visited on 2026-04-22]. Available from: <https://cocoa.altervista.org/html/partTermOrderings.html>.
50. ABBOTT, John; BIGATTI, Anna M. *CoCoALib documentation: SparsePolyRing* [online]. 2018. [visited on 2026-04-22]. Available from: <https://cocoa.altervista.org/cocoalib/doc/html/SparsePolyRing.html>.
51. ABBOTT, John; BIGATTI, Anna M. *CoCoALib documentation: GlobalManager* [online]. 2021. [visited on 2026-04-22]. Available from: <http://cocoa.altervista.org/cocoalib/doc/html/GlobalManager.html>.
52. ABBOTT, John; BIGATTI, Anna M. *CoCoALib documentation: ideal* [online]. 2022. [visited on 2026-04-22]. Available from: <http://cocoa.altervista.org/cocoalib/doc/html/ideal.html>.
53. PARISSE, Bernard; DE GRAEVE, Renée. *Xcas, Free Computer Algebra Software* [online]. 2026. [visited on 2026-04-22]. Available from: <https://xcas.univ-grenoble-alpes.fr/en.html>.
54. PARISSE, Bernard. *An Xcas Tutorial* [online]. 2026. [visited on 2026-04-22]. Available from: <https://www-fourier.univ-grenoble-alpes.fr/~parisse/giac/doc/en/tutoriel/index.html>.

- 55. DE GRAEVE, Renée; PARISSE, Bernard; BELANGER, Jay; MAROHNIC, Luka. *Xcas documentation: Command-line interface* [online]. 2007. [visited on 2026-04-22]. Available from: https://www-fourier.univ-grenoble-alpes.fr/~parisse/giac/doc/en/cascmd_en/cascmd_en002.html.
- 56. DE GRAEVE, Renée; PARISSE, Bernard; BELANGER, Jay. *Xcas documentation: Gröbner basis* [online]. 2007. [visited on 2026-04-22]. Available from: https://www-fourier.univ-grenoble-alpes.fr/~parisse/giac/doc/en/cascmd_en/cascmd_en385.html.
- 57. PARISSE, Bernard. A probabilistic and deterministic modular algorithm for computing Groebner basis over $\mathbb{Q}$. *CoRR*. 2013, vol. abs/1309.4044. Available from arXiv: 1309.4044.
- 58. ALGEBRAIC-SOLVING. *msolve* [online]. 2026. [visited on 2026-04-22]. Available from: <https://github.com/algebraic-solving/msolve>.
- 59. BERTHOMIEU, Jérémy; EDER, Christian; SAFEY EL DIN, Mohab. msolve: A Library for Solving Polynomial Systems. In: *46th International Symposium on Symbolic and Algebraic Computation*. Saint Petersburg, Russia, 2021. 46th International Symposium on Symbolic and Algebraic Computation. Available from DOI: 10.1145/3452143.3465545.
- 60. BERTHOMIEU, Jérémy; EDER, Christian; SAFEY EL DIN, Mohab. *How to Solve Multivariate Polynomial Systems with msolve?* [online]. 2026. [visited on 2026-04-22]. Available from: <https://msolve.lip6.fr/downloads/msolve-tutorial.pdf>.
- 61. GRAYSON, Daniel; STILLMAN, Michael. *Macaulay2* [online]. 2025. [visited on 2026-04-22]. Available from: <https://macaulay2.com/>.
- 62. THE MACAULAY2 AUTHORS. *Macaulay2 GitHub repository* [online]. 2025. [visited on 2026-04-22]. Available from: <https://github.com/Macaulay2/M2>.
- 63. *Macaulay2 documentation: The Macaulay2 language* [online]. [visited on 2026-04-22]. Available from: https://macaulay2.com/doc/Macaulay2/share/doc/Macaulay2/Macaulay2Doc/html/___The_sp__Macaulay2_splanguage.html.
- 64. EISENBUD, David; STILLMAN, Mike. *Macaulay2 documentation: BeginningMacaulay2* [online]. 2009. [visited on 2026-04-22]. Available from: <https://macaulay2.com/doc/Macaulay2/share/doc/Macaulay2/BeginningMacaulay2/html/>.
- 65. *Linux manual page: m2(1)* [online]. [visited on 2026-04-22]. Available from: <https://linux.die.net/man/1/m2>.

66. *Macaulay2 documentation: groebnerBasis* [online]. [visited on 2026-04-22]. Available from: https://macaulay2.com/doc/Macaulay2/share/doc/Macaulay2/Macaulay2Doc/html/_groebner__Basis.html.
67. *Macaulay2 documentation: gb* [online]. [visited on 2026-04-22]. Available from: https://macaulay2.com/doc/Macaulay2/share/doc/Macaulay2/Macaulay2Doc/html/_gb.html.
68. HINKELMANN, Franziska; STILLMAN, Mike; ARNOLD, Elizabeth. *Macaulay2 documentation: BooleanGB* [online]. 2011. [visited on 2026-04-22]. Available from: <https://macaulay2.com/doc/Macaulay2/share/doc/Macaulay2/BooleanGB/html/index.html>.
69. *Macaulay2 documentation: monomial orderings* [online]. [visited on 2026-04-22]. Available from: https://macaulay2.com/doc/Macaulay2/share/doc/Macaulay2/Macaulay2Doc/html/_monomial_sporderings.html.
70. *Macaulay2 documentation: GRevLex* [online]. [visited on 2026-04-22]. Available from: https://macaulay2.com/doc/Macaulay2/share/doc/Macaulay2/Macaulay2Doc/html/___G__Rev__Lex.html.
71. ROUNE, Bjarke Hammersholt; STILLMAN, Michael Eugene. Practical Groebner Basis Computation. *CoRR*. 2012, vol. abs/1206.6940. Available from arXiv: 1206.6940.
72. ROUNE, Bjarke Hammersholt. *mathicgb* [online]. 2012. [visited on 2026-04-22]. Available from: <https://github.com/broune/mathicgb>.
73. ROUNE, Bjarke Hammersholt. *mathic* [online]. [visited on 2026-04-22]. Available from: <https://github.com/broune/mathic>.
74. STEVENS, Marc. *M4GB: Efficient Groebner Basis algorithm* [online]. 2017. [visited on 2026-04-22]. Available from: <https://github.com/cr-marcstevens/m4gb>.
75. MAKARIM, Rusydi H.; STEVENS, Marc. M4GB: An Efficient Gröbner-Basis Algorithm. In: *Proceedings of the 2017 ACM International Symposium on Symbolic and Algebraic Computation*. Kaiserslautern, Germany: Association for Computing Machinery, 2017, pp. 293–300. ISSAC '17. ISBN 9781450350648. Available from DOI: 10.1145/3087604.3087638.
76. OPEN SOURCE INITIATIVE. *The MIT License* [online]. [visited on 2026-04-22]. Available from: <https://opensource.org/license/mit>.
77. JULIALANG CONTRIBUTORS. *Julia GitHub repository* [online]. 2026. [visited on 2026-04-22]. Available from: <https://github.com/JuliaLang/julia>.

78. JULIALANG.ORG CONTRIBUTORS. *The Julia Programming Language* [online]. 2026. [visited on 2026-04-22]. Available from: <https://julialang.org/>.
79. DEMIN, Alexander. *Groebner.jl* [online]. 2026. [visited on 2026-04-22]. Available from: <https://sumiya11.github.io/Groebner.jl/>.
80. DEMIN, Alexander. *Groebner.jl GitHub repository* [online]. 2026. [visited on 2026-04-22]. Available from: <https://github.com/sumiya11/Groebner.jl>.
81. THE JULIA LANGUAGE CONTRIBUTORS. *Julia Manual: Getting Started* [online]. 2026. [visited on 2026-04-22]. Available from: <https://docs.julialang.org/en/v1/manual/getting-started/>.
82. THE JULIA LANGUAGE CONTRIBUTORS. *Julia Manual: Command-line Interface* [online]. 2026. [visited on 2026-04-22]. Available from: <https://docs.julialang.org/en/v1/manual/command-line-interface/>.
83. *Groebner.jl documentation: Interface* [online]. 2026. [visited on 2026-04-22]. Available from: <https://sumiya11.github.io/Groebner.jl/interface/>.
84. *Groebner.jl documentation: Examples* [online]. 2026. [visited on 2026-04-22]. Available from: <https://sumiya11.github.io/Groebner.jl/examples/#Using-Low-level-interface>.
85. *OpenF4 homepage* [online]. 2016. [visited on 2026-04-23]. Available from: <https://nauotit.github.io/openf4/>.
86. LINBOX-TEAM. *Givaro: C++ Library for Arithmetic and Algebraic Computations* [online]. 2025. [visited on 2026-04-23]. Available from: <https://github.com/linbox-team/givaro>.
87. THE FFLAS-FFPACK GROUP. *FFLAS-FFPACK: Finite Field Linear Algebra Subroutines/Package* [online]. 2019. [visited on 2026-04-23]. Available from: <https://github.com/linbox-team/fflas-ffpack>.
88. *OpenF4 documentation: Tutorial for the GF(2) Finite Fields* [online]. 2016. [visited on 2026-04-23]. Available from: https://nauotit.github.io/openf4/gf2_field.html.
89. JOUX, Antoine; VITSE, Vanessa; COLADON, Titouan. *OpenF4 documentation: tutorial-gf2-method3.cpp source file* [online]. 2016. [visited on 2026-04-23]. Available from: https://nauotit.github.io/openf4/tutorial-gf2-method3_cpp_source.html.
90. CANNON, John J.; PLAYOUST, Catherine. *First Steps in Magma* [online]. 1996-08. [visited on 2026-04-23]. Available from: <https://magma.maths.usyd.edu.au/magma/pdf/first.pdf>.

91. COMPUTATIONAL ALGEBRA GROUP. *Magma Handbook: Command Line Options* [online]. 2025. [visited on 2026-04-23]. Available from: <https://magma.maths.usyd.edu.au/magma/handbook/text/47>.
92. COMPUTATIONAL ALGEBRA GROUP. *Magma Handbook: Polynomial Rings and Ideals* [online]. 2025. [visited on 2026-04-23]. Available from: <https://magma.maths.usyd.edu.au/magma/handbook/text/1312>.
93. COMPUTATIONAL ALGEBRA GROUP. *Magma Handbook: Gröbner Bases* [online]. 2026. [visited on 2026-04-23]. Available from: <https://magma.maths.usyd.edu.au/magma/handbook/text/1313>.
94. COMPUTATIONAL ALGEBRA GROUP. *Magma Handbook: Ideals and Gröbner Bases* [online]. 2025. [visited on 2026-04-23]. Available from: <https://magma.maths.usyd.edu.au/magma/handbook/text/1017>.
95. COMPUTATIONAL ALGEBRA GROUP. *Magma Handbook: Representation and Monomial Orders* [online]. 2025. [visited on 2026-04-23]. Available from: <https://magma.maths.usyd.edu.au/magma/handbook/text/1311>.
96. PYTHON SOFTWARE FOUNDATION. *Python homepage* [online]. 2026. [visited on 2026-04-26]. Available from: <https://www.python.org/>.
97. PYTHON SOFTWARE FOUNDATION. *Python documentation: asyncio* [online]. 2026. [visited on 2026-04-26]. Available from: <https://docs.python.org/3/library/asyncio.html>.
98. PYTHON SOFTWARE FOUNDATION. *Python documentation: pathlib.Path.glob* [online]. 2026. [visited on 2026-04-26]. Available from: <https://docs.python.org/3/library/pathlib.html#pathlib.Path.glob>.
99. PYTHON SOFTWARE FOUNDATION. *Python documentation: pathlib pattern language* [online]. 2026. [visited on 2026-04-26]. Available from: <https://docs.python.org/3/library/pathlib.html#pattern-language>.
100. *Linux manual page: time(1)* [online]. 2025. [visited on 2026-04-26]. Available from: <https://man7.org/linux/man-pages/man1/time.1.html>.
101. BROUWER, Andries. *The Linux kernel: Processes* [online]. 2003. [visited on 2026-04-26]. Available from: <https://aeb.win.tue.nl/linux/lk/lk-10.html>.
102. LINUX MAN-PAGES PROJECT. *Linux manual page: killpg(3)* [online]. 2025. [visited on 2026-04-26]. Available from: <https://man7.org/linux/man-pages/man3/killpg.3.html>.

103. HEO, Tejun. *Control Group v2* [online]. 2015. [visited on 2026-04-26]. Available from: <https://docs.kernel.org/admin-guide/cgroup-v2.html>.
104. SYSTEMD PROJECT. *systemd manual page: systemd* [online]. [visited on 2026-04-26]. Available from: <https://www.freedesktop.org/software/systemd/man/latest/systemd.html>.
105. SYSTEMD PROJECT. *systemd manual page: systemd-run(1)* [online]. 2026. [visited on 2026-04-26]. Available from: <https://www.freedesktop.org/software/systemd/man/latest/systemd-run.html>.
106. SYSTEMD PROJECT. *systemd manual page: systemctl* [online]. 2026. [visited on 2026-04-26]. Available from: <https://www.freedesktop.org/software/systemd/man/latest/systemctl.html>.
107. PYTHON SOFTWARE FOUNDATION. *uuid — UUID objects according to RFC 9562* [online]. 2026. [visited on 2026-04-27]. Available from: <https://docs.python.org/3/library/uuid.html>.
108. WEINER, Johannes. *PSI - Pressure Stall Information* [online]. 2018. [visited on 2026-04-27]. Available from: <https://docs.kernel.org/accounting/psi.html#psi>.
109. SYSTEMD PROJECT. *systemd manual page: journalctl* [online]. [visited on 2026-04-27]. Available from: <https://www.freedesktop.org/software/systemd/man/latest/journalctl.html>.
110. HOLÝ, Pavel. *msolve-thesis: F4 Symbolic Preprocessing Optimizations for Small-Scale AES Polynomial Systems over GF(2)* [online]. [visited on 2026-05-04]. Available from: <https://github.com/gyromean/msolve-thesis/tree/thesis>.
111. INTEL CORPORATION. *Intel Xeon Gold 6136 Processor: 24.75M Cache, 3.00 GHz* [online]. [visited on 2026-05-04]. Available from: <https://www.intel.com/content/www/us/en/products/sku/120479/intel-xeon-gold-6136-processor-24-75m-cache-3-00-ghz/specifications.html>.
112. ADVANCED MICRO DEVICES. *AMD Ryzen 7 5800X3D Gaming Processor* [online]. [visited on 2026-05-04]. Available from: <https://www.amd.com/en/products/processors/desktops/ryzen/5000-series/amd-ryzen-7-5800x3d.html>.

Contents of attached files

/	
└─ readme.md	description of the attached files
└─ license.txt	license information
└─ declaration.pdf	AI tools usage declaration
└─ benchmarking-framework	the benchmarking framework
└─ input-gen	Input generator
└─ benchmark	Benchmark runner
└─ output-parse	Output parser
└─ experimental-results.zip ..	experimental measurements and results
└─ msolve	msolve project with the proposed modifications
└─ thesis	thesis files
└─ DP_Holy_Pavel_2026.pdf	the thesis in PDF format
└─ src	the L ^A T _E X source files of the thesis