

Comparative Evaluation of Open-Source Gröbner Basis Implementations for Small-Scale AES Polynomial Systems over $\text{GF}(2)$

Pavel Holý¹ and Martin Jureček^{1*}

¹Faculty of Information Technology, Czech Technical University in
Prague, Prague, Czechia.

*Corresponding author(s). E-mail(s): martin.jurecek@fit.cvut.cz;
Contributing authors: holypav2@fit.cvut.cz;

Abstract

The practical performance of Gröbner basis implementations depends on the polynomial systems being solved. This work evaluates 20 open-source implementations with Magma as a proprietary reference on systems derived from small-scale AES over $\mathbf{GF}(2)$. The Easy, Medium, and Hard experiments use field equations and ten input instances each, comparing wall-clock runtime, peak memory, success rate, and CPU usage. Implementations with no successful runs are excluded from subsequent experiments. In Hard, msolve and Singular’s slimgb completed all ten runs successfully, with mean runtimes of 164 and 269 seconds and mean peak memory consumption of 3693 and 4343 MiB, respectively, compared with 319 seconds and 16853 MiB for Magma. During these runs, msolve used an average of 13.4 CPU cores, while slimgb and Magma used about one core each. Magma had the shortest mean runtime in Medium, whereas msolve was fastest in Easy and Hard. Changing the plaintext-ciphertext pair count affected implementations differently, and runtimes varied between instances. The results show that open-source implementations can be competitive alternatives to Magma, with their relative performance depending on the input systems.

Keywords: Gröbner basis, F4, polynomial systems, small-scale AES, algebraic cryptanalysis, open-source

1 Introduction

As more and more aspects of everyday life are being moved to the digital world, cryptosystems are becoming more important than ever. It is therefore crucial that the cryptosystems we use are robust and resilient to various attacks.

In this work, we focus on algebraic cryptanalysis of variants of AES [1]. Algebraic cryptanalysis consists of two main steps: the cryptosystem is converted into a polynomial system over a finite field, and the secret key is recovered by solving this system [2]. We use Gröbner basis computation for this purpose. Deciding whether a polynomial system has a solution is an NP-complete problem, even if all equations are quadratic and the field is $\text{GF}(2)$ [3].

In the known-plaintext attack setting considered in this work, the key is fixed but unknown, and the observed plaintext-ciphertext (PT-CT) pairs impose constraints on the same key bits. We represent these constraints by polynomial equations over $\text{GF}(2)$ using the key-variable construction of Bielik et al. [4].

By using more than one PT-CT pair, an overdefined system is obtained. Berušková et al. [5] studied reductions of such systems for Gröbner basis computation.

We focus on small-scale AES variants [6], which reduce the number of rounds and the size of the internal state while preserving an algebraic structure similar to that of AES. The variants selected for this work lead to polynomial systems that are solvable with current methods.

The contribution of this work is an experimental comparison of 20 open-source implementations for Gröbner basis computation with the commercial system Magma [7]. We evaluate polynomial systems derived from small-scale AES variants over $\text{GF}(2)$ in the Easy, Medium, and Hard experiments, with respect to runtime, memory usage, success rate, and average CPU usage. The comparison shows how performance depends on the input and identifies implementations that are competitive with Magma on the tested systems. To support this evaluation, we developed a custom benchmarking framework that generates input systems, executes implementations, and collects performance metrics.

Section 2 reviews related work on algebraic cryptanalysis of block ciphers, with particular attention to AES variants. It covers the construction and solution of polynomial systems and places our comparison of implementations in the context of earlier experiments.

Section 3 introduces the mathematical background needed to understand the experiments, including Gröbner bases and field equations over $\text{GF}(2)$. It also describes the parameters of small-scale AES and how known PT-CT pairs give rise to systems whose unknowns represent the key bits.

Section 4 describes the evaluated software systems and algorithms. It specifies the versions, polynomial rings, and monomial orderings used, and explains differences between the implementations that are relevant to the comparison.

Section 5 presents the hardware and software environment and defines the Easy, Medium, and Hard input variants. It describes the measurement conditions, resource limits, and performance metrics, including the classification of successful, timed out, and failed runs.

Section 6 presents the Easy, Medium, and Hard results, comparing the implementations in terms of runtime, memory consumption, success rate, and CPU usage. It also examines differences between input variants and individual instances, and discusses how these differences affect the interpretation of the results.

Section 7 summarizes the main findings concerning the competitiveness of open-source implementations with Magma. It states the scope of these conclusions and outlines directions for further experiments.

2 Related Work

The authors of [6] introduced small-scale variants of AES, providing a framework for comparing different cryptanalytic methods. Small-scale AES is a generalization of AES that introduces four parameters for configuring the number of rounds and the dimensions of the internal state. The authors also provided instructions for constructing systems of equations corresponding to these variants and performed an experimental evaluation using the Magma system.

Pyshkin [8] investigated the application of Gröbner bases to the cryptanalysis of block ciphers. He constructed two families of block ciphers on which he conducted experiments using Magma. For a subset of these ciphers, he presented a method for constructing zero-dimensional Gröbner bases and discussed how the approach can be applied to AES.

Bulygin and Brickenstein [9] conducted experiments on recovering the secret key of small-scale AES variants. They reported attacks on selected variants up to ten rounds, with the range of tractable variants depending on the solver and representation. They also explored a meet-in-the-middle approach, which split the polynomial system into two parts based on rounds, and made use of overdefined systems.

The authors of [10] introduced Simplified AES, a two-round algorithm designed for educational purposes. This cipher is distinct from the small-scale AES family. Its design retains resistance to trivial linear and differential cryptanalysis, allowing these methods to be studied in a simplified setting.

Further experiments were conducted in [11] on this Simplified AES cipher rather than on small-scale AES variants. The study constructed the corresponding polynomial systems and carried out algebraic cryptanalysis experiments using Magma.

The feasibility of Gröbner basis attacks on the full AES was discussed in [12]. This work proposed an algebraic cryptanalysis scheme for the full 14-round AES-256. The associated complexity analysis suggested that this approach would require fewer operations than a brute-force search. However, the proposed attack remains theoretical due to its very high complexity; this is an estimated complexity claim, not a practical key-recovery experiment.

The authors of [13] proposed a method to improve the efficiency of algebraic cryptanalysis. They take into account linear relations between intermediate state bits of the cipher and use them to simplify the resulting system of equations. Their approach makes use of the Universal Pruning technique and FWBW S-box representations to recover such linear polynomials.

Bielik et al. [4] compared the performance of CryptoMiniSat and Magma’s F4 algorithm on solving polynomial systems derived from multiple small-scale AES variants. They experimented with systems containing only key variables as well as systems including auxiliary variables. Their experiments provide a comparison between SAT solving and Gröbner basis computation in this setting.

Berušková et al. [5] explored various techniques for reducing overdefined polynomial systems. Their experimental results showed that the time required to compute a Gröbner basis for the transformed system was significantly reduced. The experiments were conducted using Magma.

The present work focuses instead on comparing a broader selection of open-source Gröbner basis implementations with Magma, using a common benchmarking framework and the Easy, Medium, and Hard inputs.

3 Background

3.1 Mathematical Background

The field $\text{GF}(2)$ consists of the elements 0 and 1, with arithmetic modulo two [14, p. 5]. Addition therefore corresponds to XOR on bits, and multiplication to logical AND. We work with polynomial systems

$$f_1(x_1, \dots, x_m) = 0, \quad \dots, \quad f_s(x_1, \dots, x_m) = 0, \quad f_i \in \text{GF}(2)[x_1, \dots, x_m],$$

where the variables represent unknown key bits.

A Gröbner basis is a generating set of an ideal whose leading terms generate the ideal of leading terms, with respect to a chosen monomial ordering [14, p. 78]. Replacing the original equations by a Gröbner basis preserves the generated ideal and hence the solution set [14, p. 31]. For a fixed ideal and monomial ordering, the reduced Gröbner basis is unique [14, pp. 93–94]; non-reduced bases need not be identical. Two orderings used in our experiments are lexicographic and degree reverse lexicographic order. These compare exponent vectors lexicographically or by total degree with a reverse-lexicographic tie break, respectively [14, pp. 56–59]. The ordering selected for each implementation is listed in Table 2.

Faugère’s F4 algorithm builds on Buchberger’s algorithm, performing polynomial reductions in batches using matrix row reduction [14, pp. 567–575]. This is the main computational idea behind several of the evaluated implementations.

The field equations

$$x_i^2 + x_i = 0, \quad 1 \leq i \leq m,$$

explicitly restrict each variable to 0 or 1. They can be supplied in the ordinary polynomial ring or imposed through the Boolean polynomial ring

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

Here the quotient-ring notation follows [14, pp. 240–246]. In this ring, $x_i^2 = x_i$, so monomials reduce to square-free monomials. This distinction in representation is relevant to the evaluated implementations.

3.2 Small-Scale AES

The evaluated Gröbner basis implementations are not specific to AES. Small-scale AES is used because it gives structured polynomial systems with adjustable parameters. The family $\text{SR}(n, r, c, e)$ [6] preserves the AES round structure while allowing smaller states and fewer rounds; its parameters are listed in Table 1.

Table 1 Parameters of $\text{SR}(n, r, c, e)$ as defined in [6].

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, $e \in \{4, 8\}$

The state is an $r \times c$ array of e -bit words, and the block and key sizes are both rce bits [6]. For example, $\text{SR}(2, 2, 2, 4)$ has a 16-bit block and key. Unlike SR^* , which omits MixColumns in the final round, SR retains it [6]. We use SR for the benchmark instances.

The input generator is based on the scripts from [15] and follows the key-variable construction described by Bielik et al. [4]. The unknowns are the $m = rce$ bits of the initial secret key; intermediate state variables are eliminated. For PT-CT pairs encrypted under the same key, we combine the corresponding systems,

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

because they share the same key solution. Each pair contributes m equations, so multiple pairs give an overdefined system. The number of pairs and the concrete data can affect solver performance, as discussed in Section 6. Deriving these systems is not the main contribution of this work; they serve as inputs to the implementation comparison.

4 Evaluated Implementations

In this comparison, an *implementation* denotes a software system or library used with a particular command or algorithm and monomial ordering. Several systems therefore contribute more than one implementation to the comparison. In total, we tested 21 implementations: 20 open-source ones and the proprietary reference system Magma. The selection includes general-purpose computer algebra systems, C/C++ libraries, and language packages; it is not an exhaustive survey. We use the implementations directly rather than through SageMath [16] interfaces, except for Sage’s own educational algorithms.

Table 2 lists the software versions, polynomial rings, and monomial orderings used in our experiments. All coefficients are in $\text{GF}(2)$. P denotes the ordinary polynomial ring and B the Boolean polynomial ring described in Section 3.1.

Table 2 Evaluated software versions, rings, and selected monomial orderings.

Software	Version	Ring	Ordering
Singular [17]	4.4.1	P	dp
SageMath [16]	10.4	P	degrevlex
BRiAl [18]	1.2.15	B	LexOrder
CoCoALib [19]	0.99880	P	DegRevLex
Giac [20]	1.9.0	P	plex, revlex
msolve [21]	0.9.4	P	degree reverse lex
Macaulay2 [22]	1.25.11	P / B	GRevLex / Lex
M4GB [23]	0a166c87	P	degree reverse lex
Groebner.jl [24]	0.10.3	P	DegRevLex
openf4 [25]	1.0.1	P	interface default
Magma [7]	V2.28-15	B	grevlex

Singular.

We tested five commands: `std`, an implementation of Buchberger’s algorithm [26]; `slimgb`, designed to keep polynomials slim [27]; `sba`, a signature-based algorithm [28]; `groebner`, which chooses a method heuristically [29]; and the Hilbert-driven `stdhilb` [30]. We used the default `sba` options and `dp`, the degree reverse lexicographic ordering recommended by the documentation [31]. Loading the user configuration was disabled.

SageMath.

We tested the educational `toy:buchberger`, `toy:buchberger2`, and `toy:d_basis` strategies of `groebner_basis` [32]. The first omits Buchberger’s criteria and does not return a reduced basis, whereas the second uses the Gebauer–Möller installation and returns a reduced basis [33]. The third implements the d -basis method [34]. We imported Sage from Python and used `degrevlex`. The result tables abbreviate these names to `sage:buchberger`, `sage:buchberger2`, and `sage:d_basis`.

BRiAl and CoCoALib.

We used BRiAl directly through its C++ library, with the default `LexOrder` of `BoolePolyRing` [35]. The computation uses `GroebnerStrategy`, `symmGB_F2`, and `minimalizeAndTailReduce`. For CoCoALib, we used `GBasis` [36] with the default `DegRevLex` ordering [37], again through the C++ library.

Giac.

We used `gbasis` [38], with both `with_cocoa` and `with_f5` set to `false`, since CoCoALib is tested separately. We tested both `plex`, the documented default [38], and `revlex`, used in [39]. We retain Giac’s ordering names without equating `revlex` to the names used by other systems.

msolve.

msolve implements F4-based Gröbner basis computation [40]. We used it to compute the full reduced Gröbner basis in degree reverse lexicographic order, as described in the msolve tutorial [41].

Macaulay2.

We tested the default `gb`, the MGB strategy based on mathicGB, and the experimental Faugère-like F4 strategy of `groebnerBasis` [42]. These three implementations use ZZ/2 and GRevLex. The fourth uses `gbBoolean` from the BooleanGB package, which works in a Boolean polynomial ring with Lex ordering [43]. The table labels are `macaulay2:gb`, `macaulay2:mgb`, `macaulay2:f4`, and `macaulay2:booleangb`.

M4GB.

M4GB extends Buchberger’s algorithm by reusing reduced polynomial multiples [44]. We used commit `0a166c87`, the M4GB solver with degree reverse lexicographic ordering, verbosity 0, field size 2, and the input’s number of variables. A binary specialized to the field size and variable count is compiled before the measured invocation.

Groebner.jl.

Groebner.jl implements F4 [45]. We used Julia 1.12.5 and the low-level coefficient/exponent-vector interface [46], with DegRevLex in `Groebner.groebner` [47]. Julia’s startup file was disabled and multithreading enabled.

openf4.

openf4 is a C++ implementation of F4 [25]. We compiled it with optimizations, vectorization, and OpenMP support, without Givaro or FFLAS-FFPACK. We adapted the GF(2) example from [48], calling `f4` and retrieving the reduced basis. Verbosity was disabled. The consulted interface does not expose a selectable monomial ordering [49].

Magma.

Magma is the proprietary reference. We used `GroebnerBasis` in a Boolean polynomial ring with grevlex; Magma uses a specialized F4 variant for this representation [50]. GPU computation was disabled.

The tested implementations differ in their polynomial rings and monomial orderings, and in whether they return a reduced Gröbner basis. These choices are described above for each implementation.

5 Experimental Setup

We created our own benchmarking framework to measure runtime, memory consumption, success rate, and CPU usage. The framework and the measured data are available in our public repository.¹ All tests were conducted on a machine running *Ubuntu*

¹<https://github.com/gyromean/groebner-aes-benchmark>

24.04.4 LTS, 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.

5.1 Inputs

Polynomial system generation is based on the work of Marek Bielik and coauthors [4, 15]. An input variant is a concrete small-scale AES variant with a defined number of PT-CT pairs; an input instance is the polynomial system generated for one key.

The experiments reported here are Easy, using $\text{SR}(2, 2, 2, 4)$ with 8 PT-CT pairs; Medium, using $\text{SR}(2, 2, 2, 4)$ with 2 pairs; and Hard, using $\text{SR}(2, 2, 4, 4)$ with 8 pairs. All reported inputs include the field equations $x_i^2 + x_i = 0$ over $\text{GF}(2)$, one for each of the $m = rce$ key variables. We tested ten instances generated from ten keys for each variant. Boolean-ring implementations impose the field relations through their ring representation.

5.2 Measurement and Metrics

The timeout was configured to 1 hour for all experiments. A memory limit of 700 GiB was enforced, with swap disabled. Memory-intensive runs were executed in isolation, giving them access to the full limit. No run exhausted the available memory because of other concurrent runs. Wherever supported, we configured parallel execution to use all 48 logical cores of the machine. Multithreaded implementations were run separately. Single-threaded implementations with moderate memory requirements could run concurrently, with up to 16 benchmark runs at a time.

Measurements cover the solver invocation, including its calling overhead and any input parsing performed within the program. Separate input and output conversion steps are excluded from the measurements. For M4GB, compilation of the input-specialized binary takes place before measurement and is not included in the reported runtime.

In the summary tables, OK requires successful program completion and a correct computed Gröbner basis. TO denotes termination at the one-hour timeout, and Fail denotes other unsuccessful runs, including nonzero return codes and processes killed for excessive memory use. Success rate is the fraction of the ten input instances for which the run is classified as OK.

Time is the wall-clock runtime in seconds. Memory is the peak memory consumption of the run and its subprocesses in mebibytes ($1 \text{ MiB} = 2^{20} \text{ B}$). CPU is the average number of CPU cores used during the run, calculated by dividing cumulative CPU time by wall-clock time. Time, Memory, and CPU are reported as arithmetic means over successful runs only.

Memory and CPU measurements are collected at one-second intervals. Each sample retains earlier memory peaks and accumulated CPU usage, but updates after the last sample, during approximately the final second of a run, may be missed. This mainly affects runs with durations close to the sampling interval.

6 Experimental Results

In this section we use the benchmarking framework to evaluate the implementations from Section 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.

Adding more PT-CT pairs adds equations that share the same unknown key, making the system more overdefined. Easy and Medium use the same small-scale AES variant with different pair counts, while Hard uses a larger state. The experiment names do not imply increasing difficulty for every implementation.

The three experiments are defined in Section 5. Easy evaluates all 21 implementations, while Medium and Hard test only those with at least one successful run in the preceding experiment, leaving 16 and 15 implementations, respectively. Each experiment uses ten generated input instances. Implementations omitted from a later table were not tested in that experiment because they had no successful runs in the preceding one.

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 5.2. Averages over successful runs must be read together with the success counts, particularly for implementations that solved only a small fraction of the inputs.

The runtime plots retain all tested implementations and individual runs, including unsuccessful ones. Points may overlap. Circles mark successful runs. Crosses show how long a run lasted before it timed out or failed. The memory plot for Hard additionally shows the spread of peak memory and the distinction between successful and unsuccessful runs. Memory means for all three experiments remain in the tables.

6.1 Easy

This experiment uses SR(2, 2, 2, 4) with 8 PT-CT pairs. The results are shown in Table 3 and Figure 1.

Fifteen implementations completed all ten runs successfully. BRiAl completed eight; `singular:stdhilb`, `singular:sba`, `macaulay2:booleangb`, and `sage:buchberger` timed out on all ten, while `openf4` failed on all ten. The shortest mean runtime was 0.93 seconds for `msolve`, followed by 1.30 seconds for `singular:slimgb`, compared with 11.3 seconds for Magma. Their mean peak memory values were 6.78, 55.5, and 2487 MiB, respectively. At runtimes close to or below the one-second sampling interval, memory and CPU measurements may miss a substantial part of execution; these short-run resource values should therefore be interpreted cautiously.

Table 3 Summary of the Easy experiment.

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

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

6.2 Medium

This experiment uses $\text{SR}(2, 2, 2, 4)$ with 2 PT-CT pairs. The results are shown in Table 4 and Figure 2.

Twelve implementations completed all ten runs successfully. `BRiAl` and `macaulay2:gb` each completed two runs and timed out on eight; `sage:d_basis` completed one and timed out on nine. Only `sage:buchberger2` timed out on every instance. Magma had the shortest mean runtime, 3.55 seconds, followed by `msolve` at 4.02 seconds and `singular:slimgb` at 7.65 seconds. `BRiAl` and `macaulay2:gb` averaged 844 and 2880 seconds over their two successful runs, respectively, while the single successful run of `sage:d_basis` took 2352 seconds.

The largest mean peak memory consumption over successful runs was 3278 MiB for `M4GB`. The mean CPU usage reached 33.5 cores for `msolve`, while the other implementations averaged at most 4.20 cores over successful runs.

6.3 Hard

This last experiment uses $\text{SR}(2, 2, 4, 4)$ with 8 PT-CT pairs. The results are shown in Table 5 and in Figures 3 and 4.

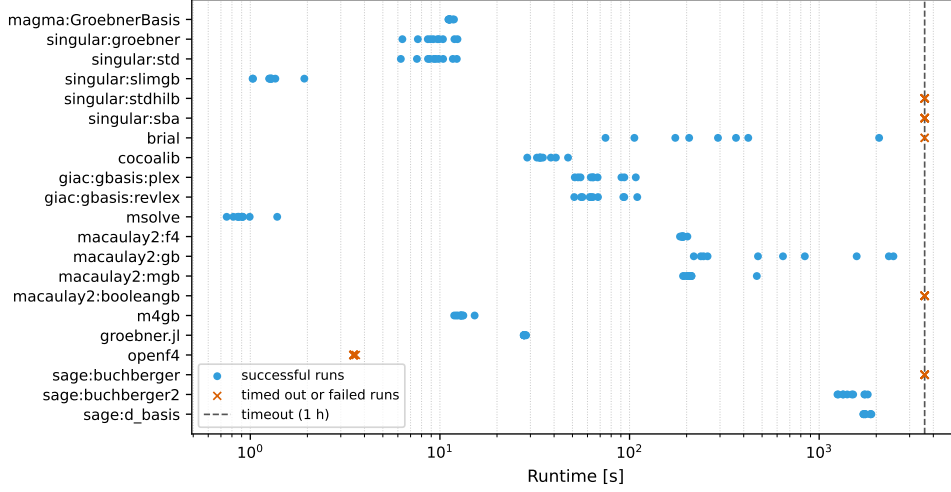


Fig. 1 Runtime of the Easy experiment, all 21 implementations. Each point represents one run for one key; overlapping points are possible. The dashed line marks the one-hour timeout.

Table 4 Summary of the Medium experiment.

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

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

Six open-source implementations completed all ten runs: `singular:slimgb`, both Giac implementations, `msolve`, `macaulay2:f4`, and `macaulay2:mgb`. Magma also completed all ten. Seven other implementations timed out on all ten inputs, while `groebner.jl` failed on all ten. Among successful implementations, the longest mean

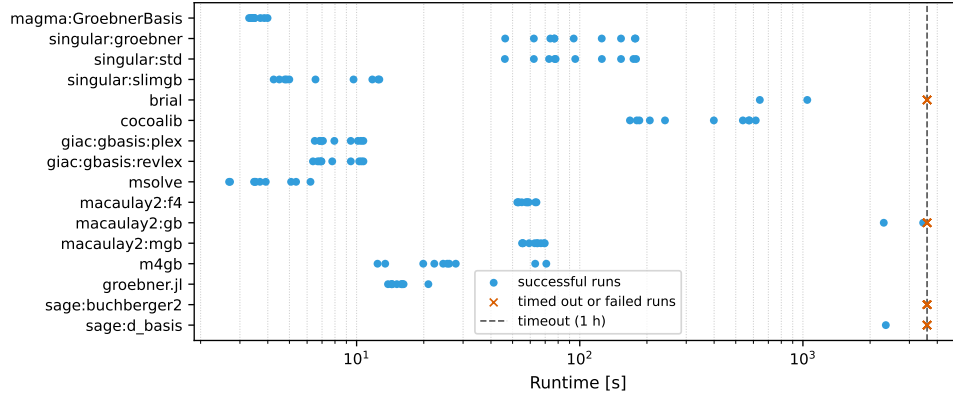


Fig. 2 Runtime of the Medium experiment, all 16 tested implementations. Each point represents one run for one key; the dashed line marks the one-hour timeout.

Table 5 Summary of the Hard experiment.

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

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

runtimes were 2713 and 2544 seconds for the two Macaulay2 implementations. The two Giac variants averaged 725 and 726 seconds. `singular:slimgb` and `msolve` took 269 and 164 seconds, respectively, both below the 319 seconds of Magma. The shortest mean wall-clock runtime in this experiment was therefore achieved by `msolve`.

The mean peak memory consumption over successful runs was 34524 MiB for `macaulay2:f4`, 16853 MiB for Magma, and 17931 MiB for `macaulay2:mgb`. The remaining successful implementations averaged at most 5206 MiB, including 3693 MiB

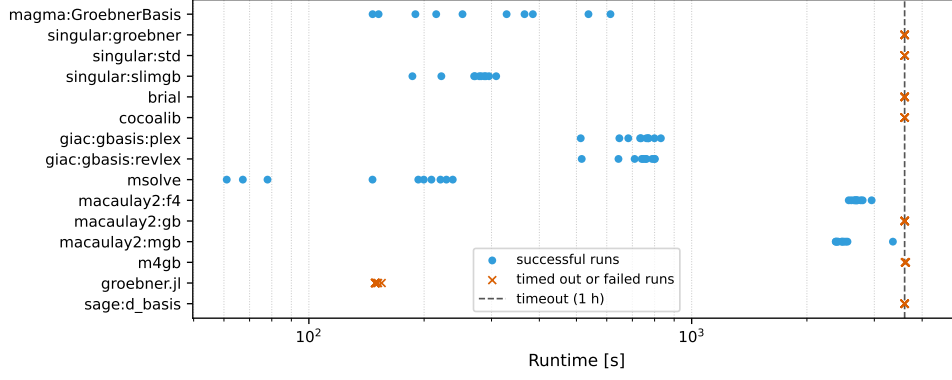


Fig. 3 Runtime of the Hard experiment, all 15 tested implementations. Each point represents one run for one key; the dashed line marks the one-hour timeout. Failed runs are not successful computations with short runtimes.

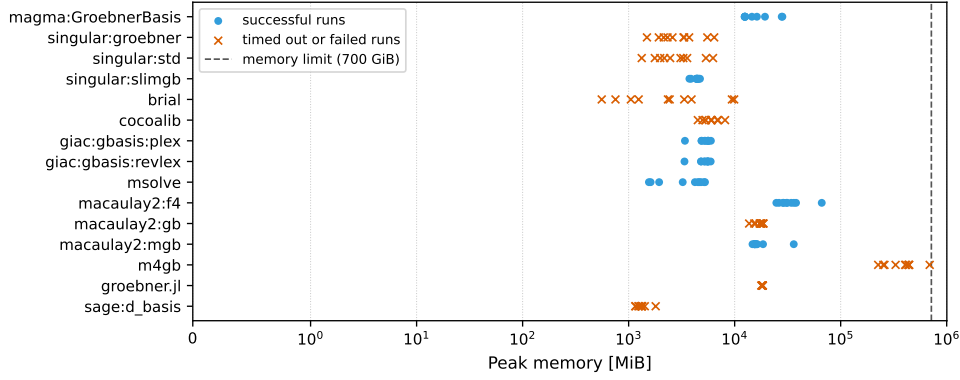


Fig. 4 Peak memory consumption of the Hard experiment, all 15 tested implementations. Each point represents one run for one key, including unsuccessful runs. The axis is linear from 0 to 1 MiB and logarithmic above 1 MiB. The dashed line marks the memory limit (700 GiB); memory-intensive runs had access to the full limit.

for `msolve` and 4343 MiB for `singular:slimgb`. Both thus had lower mean peak memory consumption than Magma. During these runs, `msolve` used an average of 13.4 CPU cores, while `singular:slimgb` and Magma used about one core each.

6.4 Input Sensitivity

The tested implementations are data-sensitive. Comparing Tables 3 and 4, increasing the PT-CT pair count from 2 to 8 corresponds to a decrease in mean runtime from 4.02 to 0.93 seconds for `msolve` and from 7.65 to 1.30 seconds for `singular:slimgb`, but an increase from 3.55 to 11.3 seconds for Magma and from 62.3 to 231 seconds for

`macaulay2:mgb`. All four implementations succeeded on all ten inputs in both experiments, so these contrasts are not caused by excluding timed-out runs from the means. More pairs do not make the computation uniformly easier across implementations.

Runtime also differs between instances of the same input variant. In Medium, BRiAl completed two runs in approximately 641 and 1046 seconds, while the remaining eight exceeded the one-hour timeout (Figure 2). In Hard, successful runtimes ranged from approximately 61 to 238 seconds for `msolve` and from 147 to 614 seconds for Magma (Figure 3). The means alone therefore do not describe every key’s runtime.

For other input variants, the results could be different. These ten-instance experiments show that several open-source implementations are competitive on the selected inputs, with their relative performance depending on the input variant. The success counts and measured CPU usage provide further context for the runtime and memory averages.

7 Conclusion

This work evaluated 20 open-source implementations for Gröbner basis computation using polynomial systems derived from small-scale AES instances over $\text{GF}(2)$, with Magma as a proprietary reference. The Easy, Medium, and Hard experiments used field equations and compared wall-clock runtime, peak memory, success rate, and CPU usage. Only implementations with at least one successful run proceeded to the next experiment.

In Hard, `msolve` and `singular:slimgb` both completed all ten runs with lower mean wall-clock runtime and peak memory than Magma: 164 seconds and 3693 MiB for `msolve`, 269 seconds and 4343 MiB for `singular:slimgb`, and 319 seconds and 16853 MiB for Magma. During these runs, `msolve` used an average of 13.4 CPU cores, while `singular:slimgb` and Magma used about one core each (Table 5).

The shortest mean runtime was achieved by `msolve` in Easy and by Magma in Medium. Runtimes are strongly affected by the inputs; even changing the PT-CT pair count has different effects across implementations. Successful-run means must be read together with success counts. On the tested inputs, several open-source implementations were competitive alternatives to Magma, particularly `msolve` and `singular:slimgb` in Hard. Future work could extend the comparison to larger small-scale AES variants and polynomial systems derived from other ciphers.

Acknowledgements

This work was supported by the Student Summer Research Program 2025 of FIT CTU in Prague.

References

- [1] National Institute of Standards and Technology: Advanced encryption standard (AES). Federal Information Processing Standards Publication NIST FIPS 197-upd1, National Institute of Standards and Technology (May 2023). <https://doi.org/10.6028/NIST.FIPS.197-upd1>
- [2] Bard, G.V.: Algebraic Cryptanalysis. Springer, New York, NY (2009). <https://doi.org/10.1007/978-0-387-88757-9>
- [3] Courtois, N., Klimov, A., Patarin, J., Shamir, A.: Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In: Advances in Cryptology — EUROCRYPT 2000. Lecture Notes in Computer Science, pp. 392–407. Springer, Berlin, Heidelberg (2000). https://doi.org/10.1007/3-540-45539-6_27
- [4] Bielik, M., Jureček, M., Jurečková, O., Lórencz, R.: Yet another algebraic cryptanalysis of small scale variants of AES. In: Proceedings of the 19th International Conference on Security and Cryptography – SECRYPT, pp. 415–427. SCITEPRESS – Science and Technology Publications, Setúbal, Portugal (2022). <https://doi.org/10.5220/0011327900003283>
- [5] Berušková, J., Jureček, M., Jurečková, O.: 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 **20**(4), 885–900 (2024) <https://doi.org/10.1007/s11416-024-00540-2>
- [6] Cid, C., Murphy, S., Robshaw, M.J.B.: Small scale variants of the AES. In: Fast Software Encryption, vol. 3557, pp. 145–162. Springer, Berlin, Heidelberg (2005). https://doi.org/10.1007/11502760_10
- [7] Computational Algebra Group: Magma Computational Algebra System (2026). <https://magma.maths.usyd.edu.au/magma/> Accessed 2026-04-21
- [8] Pyshkin, A.: Algebraic cryptanalysis of block ciphers using Groebner bases. PhD thesis, Technische Universität Darmstadt, Darmstadt, Germany (2008). <https://nbn-resolving.org/urn:nbn:de:tuda-tuprints-10608>
- [9] Bulygin, S., Brickenstein, M.: Obtaining and solving systems of equations in key variables only for the small variants of AES. Mathematics in Computer Science **3**(2), 185–200 (2010) <https://doi.org/10.1007/s11786-009-0020-y>
- [10] Musa, M.A., Schaefer, E.F., Wedig, S.: A simplified AES algorithm and its linear and differential cryptanalyses. Cryptologia **27**(2), 148–177 (2003) <https://doi.org/10.1080/0161-110391891838>
- [11] Simmons, S.: Algebraic cryptanalysis of simplified AES. Cryptologia **33**(4), 305–314 (2009) <https://doi.org/10.1080/01611190903185328>

- [12] Zhao, K., Cui, J., Xie, Z.: Algebraic cryptanalysis scheme of AES-256 using Gröbner basis. *Journal of Electrical and Computer Engineering* **2017**, 1–9 (2017) <https://doi.org/10.1155/2017/9828967>
- [13] Arabnezhad, H., Sadeghian, B.: Toward a more efficient Gröbner-based algebraic cryptanalysis. *Journal of Computing and Security* **7**(2), 103–117 (2020) <https://doi.org/10.22108/jcs.2020.123673.1050>
- [14] Cox, D.A., Little, J., O’Shea, D.: *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra*, 5th edn. Undergraduate Texts in Mathematics. Springer, Cham, Switzerland (2025). <https://doi.org/10.1007/978-3-031-91841-4>
- [15] Bielik, M.: YAAC of AES: experiments (2020). <https://gitlab.com/upbqdn/yaac/-/tree/main/experiments> Accessed 2026-04-26
- [16] SageMath: SageMath - Open-Source Mathematical Software System. <https://www.sagemath.org/> Accessed 2026-04-21
- [17] Decker, W., Greuel, G.-M., Pfister, G., Schönemann, H.: Singular. <https://www.singular.uni-kl.de/> Accessed 2026-04-21
- [18] BRiAl: BRiAl GitHub repository (2025). <https://github.com/BRiAl/BRiAl> Accessed 2026-04-21
- [19] cocoa-official: CoCoALib (2026). <https://github.com/cocoa-official/CoCoALib> Accessed 2026-04-22
- [20] Parisse, B., De Graeve, R.: Xcas, Free Computer Algebra Software (2026). <https://xcas.univ-grenoble-alpes.fr/en.html> Accessed 2026-04-22
- [21] algebraic-solving: msolve (2026). <https://github.com/algebraic-solving/msolve> Accessed 2026-04-22
- [22] The Macaulay2 Authors: Macaulay2 GitHub repository (2025). <https://github.com/Macaulay2/M2> Accessed 2026-04-22
- [23] Stevens, M.: M4GB: Efficient Groebner Basis algorithm (2017). <https://github.com/cr-marcstevens/m4gb> Accessed 2026-04-22
- [24] Demin, A.: Groebner.jl GitHub repository (2026). <https://github.com/sumiya11/Groebner.jl> Accessed 2026-04-22
- [25] OpenF4 homepage (2016). <https://nauotit.github.io/openf4/> Accessed 2026-04-23
- [26] Singular Manual: std (2025). <https://www.singular.uni-kl.de/hannes/sing.364.htm#SEC404> Accessed 2026-04-21

- [27] Singular Manual: slimgb (2025). https://www.singular.uni-kl.de/hannes/sing_358.htm#SEC398 Accessed 2026-04-21
- [28] Singular Manual: sba (2025). https://www.singular.uni-kl.de/hannes/sing_353.htm#SEC393 Accessed 2026-04-21
- [29] Singular Manual: groebner (2025). https://www.singular.uni-kl.de/hannes/sing_266.htm#SEC306 Accessed 2026-04-21
- [30] Singular Manual: stdhilb (2022). https://www.singular.uni-kl.de/Manual/4-3-1/sing_359.htm Accessed 2026-04-21
- [31] Singular Manual: Term orderings (2016). https://www.singular.uni-kl.de/Manual/4-0-3/sing_31.htm Accessed 2026-04-21
- [32] The Sage Development Team: Sage Reference Manual: MPolynomialIdeal.groebner.basis (2025). 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 Accessed 2026-04-21
- [33] Albrecht, M., Hampton, M., The Sage Development Team: Sage Reference Manual: Educational Versions of Groebner Basis Algorithms (2025). https://doc.sagemath.org/html/en/reference/polynomial_rings/sage/rings/polynomial/toy_buchberger.html#sage.rings.polynomial.toy_buchberger Accessed 2026-04-21
- [34] Albrecht, M.: Sage Reference Manual: Educational Version of the d -Groebner Basis Algorithm over PIDs (2025). https://doc.sagemath.org/html/en/reference/polynomial_rings/sage/rings/polynomial/toy_d_basis.html Accessed 2026-04-21
- [35] Dreyer, A.: BRiAl GitHub repository: BoolePolyRing.h (2006). <https://github.com/BRiAl/BRiAl/blob/master/libbrial/include/polybori/BoolePolyRing.h> Accessed 2026-04-21
- [36] Abbott, J., Bigatti, A.M.: CoCoALib documentation: ideal (2022). <http://cocoa.altervista.org/cocoalib/doc/html/ideal.html> Accessed 2026-04-22
- [37] CoCoA Manual: Term Orderings. <https://cocoa.altervista.org/html/partTermOrderings.html> Accessed 2026-04-22
- [38] De Graeve, R., Parris, B., Belanger, J.: Xcas documentation: Gröbner basis (2007). https://www-fourier.univ-grenoble-alpes.fr/%7Eparisse/giac/doc/en/cascmd_en/cascmd_en385.html Accessed 2026-04-22
- [39] Parris, B.: A probabilistic and deterministic modular algorithm for computing Groebner basis over $\mathbb{Q}$. CoRR **abs/1309.4044** (2013) [1309.4044](https://arxiv.org/abs/1309.4044)

- [40] Berthomieu, J., Eder, C., Safey El Din, M.: msolve: A Library for Solving Polynomial Systems. In: Proceedings of the 2021 International Symposium on Symbolic and Algebraic Computation, pp. 51–58. ACM, New York, NY, USA (2021). <https://doi.org/10.1145/3452143.3465545> . <https://hal.sorbonne-universite.fr/hal-03191666>
- [41] Berthomieu, J., Eder, C., Safey El Din, M.: How to Solve Multivariate Polynomial Systems with msolve? (2026). <https://msolve.lip6.fr/downloads/msolve-tutorial.pdf> Accessed 2026-04-22
- [42] Macaulay2 documentation: groebnerBasis. https://macaulay2.com/doc/Macaulay2/share/doc/Macaulay2/Macaulay2Doc/html/_groebner__Basis.html Accessed 2026-04-22
- [43] Hinkelmann, F., Stillman, M., Arnold, E.: Macaulay2 documentation: BooleanGB (2011). <https://macaulay2.com/doc/Macaulay2/share/doc/Macaulay2/BooleanGB/html/index.html> Accessed 2026-04-22
- [44] Makarim, R.H., Stevens, M.: M4GB: An efficient Gröbner-basis algorithm. In: Proceedings of the 2017 ACM International Symposium on Symbolic and Algebraic Computation. ISSAC '17, pp. 293–300. Association for Computing Machinery, New York, NY, USA (2017). <https://doi.org/10.1145/3087604.3087638>
- [45] Demin, A.: Groebner.jl (2026). <https://sumiya11.github.io/Groebner.jl/> Accessed 2026-04-22
- [46] Groebner.jl documentation: Examples (2026). <https://sumiya11.github.io/Groebner.jl/examples/#Using-Low-level-interface> Accessed 2026-04-22
- [47] Groebner.jl documentation: Interface (2026). <https://sumiya11.github.io/Groebner.jl/interface/> Accessed 2026-04-22
- [48] Joux, A., Vitse, V., Coladon, T.: OpenF4 documentation: tutorial-gf2-method3.cpp source file (2016). https://nauotit.github.io/openf4/tutorial-gf2-method3.8cpp_source.html Accessed 2026-04-23
- [49] OpenF4 documentation: Tutorial for the GF(2) Finite Fields (2016). https://nauotit.github.io/openf4/gf2_field.html Accessed 2026-04-23
- [50] Computational Algebra Group: Magma Handbook: Gröbner Bases (2026). <https://magma.maths.usyd.edu.au/magma/handbook/text/1314> Accessed 2026-09-06