Eight Implementations, One Cipher: A Cross-Language Differential Study of SM4 (GB/T 32907-2016)

Daozhendong Yang
School of Intelligent Science and Engineering,
Yunnan Technology and Business University, Kunming, China
hi@yxpil.com
https://github.com/yxpil/SM4Kit

Abstract

SM4 is a 128-bit block cipher standardised in GB/T 32907-2016. SM4Kit is an open-source library that implements it eight times — in JavaScript, Python, Go, Java, Rust, PHP, Ruby and C — behind one API, with no third-party dependencies. This paper is not a proposal for a cipher and claims no cryptographic novelty; it is a differential-testing study of those eight implementations. We describe a harness that drives each of them through the GB/T 32907 Annex A vectors, a deterministic corpus of 256 random keys and blocks, and nine ECB/CBC payloads at boundary lengths, and that compares ciphertexts, the full 32-word round-key schedule, and mode outputs against one another and against a harness-local reference model. Four toolchains (JavaScript, Python, Java, Rust) were available on the authoring machine and were executed; Go, PHP, Ruby and C were read and statically checked but *not executed*, and are reported as not executed rather than as passes. For the four executed implementations the harness finds no divergence at all: 259 block cases, 8,288 round-key words and 18 mode outputs per language agree exactly, and all six language pairs agree pairwise. A static extraction of the S-box table from all eight sources finds a single 256-entry bijection, identical everywhere. To show that this null result is not vacuous we inject 294 single-point mutants into the reference model: the standard vector alone detects 244 of them (83.0%), missing 50 of the 256 S-box table mutations, while a 16-case differential corpus detects 294 of 294. The harness, corpus, drivers, raw transcripts and figure scripts are released so that a reader with the four missing toolchains can complete the matrix.

Artifact. The library, its eight self-tests, the differential harness, the mutation study and the figure scripts are open source at https://github.com/yxpil/SM4Kit. The paper's own scripts, corpora, raw driver transcripts and result files are included with this submission.

1 Introduction

SM4 is the block cipher specified in the Chinese national standard GB/T 32907-2016 Standardization Administration of the People's Republic of China (2016). It operates on 128-bit blocks under a 128-bit key, runs 32 rounds, and uses an 8-bit S-box followed by a linear transform. The algorithm is public, thoroughly specified, and small enough to be reimplemented in an afternoon: the whole cipher is a table of 256 bytes, four constants, a formula for 32 round constants, and a 32-iteration loop.

SM4Kit is a library that does exactly that, eight times. It ships one file per language — JavaScript, Python, Go, Java, Rust, PHP, Ruby and C — each with no third-party dependency, the same four-function API (ECB and CBC, each with PKCS#7 padding), the same error discipline within each language's idiom, and an in-file self-test against the two examples of GB/T 32907 Annex A.

That redundancy is the subject of this paper. A cipher implemented eight times is a natural experiment in *convention drift*: the same mathematics expressed through eight different integer models, eight different byte-buffer types, and eight copies of a 256-entry constant table. The standard test vector is a single point in the input space. It will not notice a mistyped S-box entry that the vector's words never index, and it cannot notice that two implementations disagree with each other while both pass. This paper asks a narrow software-engineering question: *if one standard is implemented eight times, do the implementations agree, and what does it take to find out?*

We answer it with a differential harness rather than with a new cipher. The harness drives every implementation through the standard vector, a seeded random corpus, and a set of boundary-length payloads; it compares every ciphertext, every one of the 32 round keys, and every mode output against a reference model and against every other implementation; and it records, per language, whether the language was actually executed or merely read. The last point is not bookkeeping. Four of the eight toolchains were not installed on the authoring machine, and this paper reports those four as *not executed* everywhere, including in the headline matrix, rather than quietly omitting them or inferring a pass from a code review.

1.1 Contributions

C1. A cross-language differential harness for one standard (§3). A uniform tab-separated job protocol and one driver per language, each calling only the implementation's pub-

lic API — no driver reimplements SM4. The harness compares three classes of observable: the 32-byte ECB image of a 16-byte block (which is two block encryptions, because padding appends a second block), all 32 round keys, and full ECB/CBC outputs for payloads of length 0 to 255 bytes.

C2. An execution-status discipline (§3.4). Every cell of the result matrix carries one of *executed*, *not executed* or *driver failed*, plus a machine-readable reason. A language whose toolchain is absent is never reported as passing. Four languages were executed here; four were read only.

C3. A measured null result on the executed set (§5). Across JavaScript, Python, Java and Rust the harness finds zero divergences: 259 block ciphertexts, 8,288 round-key words and 18 mode outputs per language, and zero differences in all six language pairs. All four also reproduce the standard vector exactly. We report the null result as the result it is, and §5.2 states precisely what it does not establish.

C4. A static table-conformance check over all eight sources (§5.3). A parser extracts the 256-entry S-box from each source file, checks that it is a bijection, and compares the eight tables to each other. This is the only evidence the paper offers about Go, PHP, Ruby and C, and we label it as such.

C5. A mutation study that calibrates the corpus (§5.4). We inject 294 single-point faults into the harness's reference model and ask which of them each test corpus detects. The single standard vector detects 244 of 294 (83.0%) and misses 50 of the 256 S-box table mutations; a 4-case differential corpus detects 291 (99.0%), and a 16-case corpus detects all 294. This is the quantitative argument for differential testing over vector-only testing, and it is measured on the same cipher the rest of the paper studies.

1.2 Non-goals

We do not propose a new cipher, modify SM4, or claim any cryptanalytic result. SM4 is a standard Standardization Administration of the People's Republic of China (2016); the novelty here is the conformance methodology and the released artefacts, not the algorithm.

We do not claim that the four implementations we could not execute are correct. We claim only that their source passes the static checks of §5.3, and we label them "not executed" in every table and figure.

We do not claim side-channel resistance. All eight implementations index the S-box with secret-dependent data, as the straightforward formulation of the cipher does; cache-timing behaviour is out of scope and is not measured here Kocher (1996).

We do not benchmark performance, and we do not claim production readiness. The library implements confidentiality only: ECB and CBC with PKCS#7, no MAC and no AEAD, which the repository's own README states plainly.

Finally, we do not claim that agreement implies correctness. Agreement among implementations that were all written by the same author from the same standard rules out divergence, not shared error. §6 makes this concrete with a counted blind spot.

2 Background

2.1 The cipher

SM4 encrypts a 128-bit block under a 128-bit key in 32 rounds. Write a block as four 32-bit words (X_0, X_1, X_2, X_3), each the big-endian concatenation of four bytes. For $i = 0, \ldots, 31$,

$$X_{i+4} = X_i \oplus T(X_{i+1} \oplus X_{i+2} \oplus X_{i+3} \oplus rk_i)\,, \qquad (1)$$

where T applies the 8-bit S-box S to each of the four bytes and then the linear transform L:

$$\begin{aligned} T(x) &= L(\tau(x)), \\ L(B) &= B \oplus \mathrm{ROTL}(B,2) \oplus \mathrm{ROTL}(B,10) \\ &\quad \oplus \mathrm{ROTL}(B,18) \oplus \mathrm{ROTL}(B,24). \end{aligned} \qquad (2)$$

The ciphertext is the reverse-ordered tuple $(X_{35}, X_{34}, X_{33}, X_{32})$. The key schedule is the same shape with a different linear transform:

$$rk_i = K_i \oplus T'(K_{i+1} \oplus K_{i+2} \oplus K_{i+3} \oplus CK_i)\,, \qquad (3)$$

$$\begin{aligned} L'(B) &= B \oplus \mathrm{ROTL}(B,13) \oplus \mathrm{ROTL}(B,23), \\ ck_{i,j} &= (4i + j) \cdot 7 \bmod 256, \end{aligned} \qquad (4)$$

with $(K_0, K_1, K_2, K_3) = (MK_0 \oplus FK_0, \ldots, MK_3 \oplus FK_3)$ and $FK = \{\texttt{A3B1BAC6}, \texttt{56AA3350}, \texttt{677D9197}, \texttt{B27022DC}\}$.

Two structural facts matter for everything that follows. First, decryption is the same round function with the round keys in reverse order, so every implementation's decrypt path is "reverse the key schedule and rerun" — a place where an off-by-one is easy and, as §5.4 shows, easy to detect. Second, the only intermediate state the standard exposes is the round key sequence $rk_0, \ldots, rk_{31}$. The round registers X_i are internal, and none of the eight implementations makes them public. The harness therefore compares round keys, not round registers; §6 treats this as a limitation rather than pretending otherwise.

2.2 Modes and padding

The library provides ECB and CBC, each with PKCS#7 padding on encryption and padding validation on decryption Kaliski (1998); Dworkin (2001). ECB encrypts each block independently; CBC XORs each plaintext block with the previous ciphertext block, or with the IV for the first block. Both are confidentiality-only constructions. CBC with PKCS#7 and no MAC is unauthenticated encryption and is exposed to padding-oracle attacks if an attacker can observe decryption failures; the repository says so in its security notes, and this paper does not test that surface.

2.3 Why separate implementations drift

Five mechanisms produce divergence between implementations that all claim to implement the same specification.

Byte order at the word boundary. The standard's words are big-endian byte sequences. An implementation that loads a word with a native integer read, or writes one with `memcpy`, inherits the host's byte order Cohen (1981). Every one of the eight implementations here avoids that by assembling words byte by byte, which is the right decision and also the reason the harness finds no endianness fault: the convention is written out in all eight sources.

Signedness and word width. JavaScript's bitwise operators produce *signed* 32-bit values, so every shift must be followed by `>>>0`. Java's `byte` is signed, so every byte read needs `& 255`. Java's `int` is signed but `>>>` is available. PHP integers are 64-bit, so a left shift silently escapes the 32-bit word unless the result is masked. Python and Ruby integers are unbounded and need the same mask. C promotes `uint8_t` to `int`, and a left shift into the sign bit of a signed type is undefined behaviour unless the operand is unsigned. Each of these is a place where a correct-looking expression is wrong for exactly one language.

Table transcription. Every implementation embeds the same 256 hex constants. A single mistyped nibble produces a program that compiles, runs, and passes any test that does not index that entry.

Constant generation. CK_i can be precomputed at load time or derived per round. Both are correct; a typo in the derivation affects all 32 rounds at once, while a typo in a precomputed table affects one round.

Test-vector reach. The standard vector fixes one key and one plaintext. During its 32 rounds and 32 key-schedule rounds the S-box is indexed at most $32 \times 8 = 256$ byte positions, and §5.4 measures that this reaches 206 of the 256 table entries. The other 50 entries are untested by the vector, no matter how carefully the vector is checked.

2.4 Differential testing

Differential testing executes several implementations of the same specification on the same input and treats a disagreement as a failure McKeeman (1998). It has been used to find compiler bugs at scale Yang et al. (2011) and, more recently, to test database engines Rigger and Su (2020). It is closely related to N-version programming Avižienis (1985) in structure — several versions, one specification — but its purpose is inverted: N-version programming runs versions in parallel to tolerate a fault, whereas differential testing runs them side by side to expose one. The oracle problem — deciding what the right answer is when no version is trusted Barr et al. (2015) — is addressed here by anchoring on the standard vector and by comparing implementations with each other rather than only with a model. Randomised input generation in the style of QuickCheck Claessen and Hughes (2000) and the fuzzing literature Manès et al. (2019) supplies the corpus.

For cryptographic code specifically, three families of prior artefact exist: validation suites such as the NIST CAVP/ACVP programme National Institute of Standards and Technology, curated edge-case vector sets such as Wycheproof Google, and cross-*library* differential fuzzers such as Cryptofuzz Vranken. All three test many implementations of a primitive, but almost always within one language and one build system, and their reports are about libraries. The study here keeps the primitive fixed and varies the *language*, which is the axis on which integer semantics, buffer types and table representation differ.

3 Method

3.1 What is compared

For every block case the harness records three observables per language:

(i) the full 32-byte ECB encryption of the 16-byte block. Because the API pads, a 16-byte input becomes two blocks: the block itself, and the block XORed with sixteen `0x10` bytes. One call therefore yields two independent block encryptions and a check on the padding path.

(ii) the 32 round keys returned by the key-schedule function, formatted as eight hex digits each. This is the only intermediate state the API exposes.

(iii) for the payload corpus, the complete ECB and CBC ciphertexts.

The reference model is a ninth implementation, written for this paper in Python (`scripts/reference.py`) directly from the equations of §2, and kept out of the library under test. It is anchored to the standard by the Annex A vector, which it reproduces. It is not treated as ground truth: the paper reports both model-versus-implementation and implementation-versus-implementation comparisons, and all three agree.

3.2 Corpus

The block corpus has 259 cases: the GB/T 32907 Annex A example 1 key and plaintext; an all-zero key and block; an all-`0xFF` key and block; and 256 pseudo-random (key, block) pairs. The random cases come from a Mersenne Twister seeded with `0x534D34` (5,459,252), so the corpus is byte for byte reproducible by anyone who runs the harness; the generated corpus is also committed as `data/jobs/cases.tsv`.

The payload corpus has nine cases with lengths $0, 1, 15, 16, 17, 31, 32, 100, 255$ bytes, each with its own random key and IV. The lengths straddle the block boundary in both directions and include a multi-block message; PKCS#7 always appends at least one byte, so the 16-byte and 32-byte payloads exercise the "full block of padding" path.

3.3 Driver protocol

Each language has a driver under `scripts/drivers/` that reads two tab-separated files and writes a third. Records are `BLOCK` (id, ciphertext, round keys), `ECB` (id, ciphertext) and `CBC` (id, ciphertext); there is no header row, so every driver can parse the corpus with one string split. Drivers call the implementation's public functions only. JavaScript, Python, Java, Rust, PHP and Ruby drivers call `ecbEncrypt`/`cbcEncrypt`/`roundKeys` (or the local spelling); the C driver calls the four functions declared in `sm4.h`.

Two staging details are worth recording because they are the only places where the harness touches implementation files. The Rust driver is compiled with `#[path]` against a copy of `src/lib.rs`, so that the library becomes a module of the driver; no line of the library is edited. The Go file is `package main` with a demo `func main()`, so the harness copies it and renames that one function to `repoDemoMain` to avoid a duplicate symbol; again the implementation body is untouched.

3.4 Execution status

A test matrix is only as honest as its treatment of the cells it could not fill. The harness therefore assigns every language one of three statuses — *executed*, *not_executed*, *driver_failed* — and stores the reason. Toolchain discovery is a `PATH` lookup for the interpreter or compiler named in the repository's own README. If the lookup fails, the language is marked not executed, no driver is run, and the matrix cell is rendered grey and labelled "not run" (Figure 2). The same status propagates to the CSV and to the JSON, so a reader who parses `data/differential.json` cannot mistake an absent measurement for a passing one.

3.5 In-repository self-tests

Independently of the differential corpus, the harness runs each implementation's own self-test, with the repository's `-long` flag where it exists, and parses the reported result of the standard vector (KAT1), the million-iteration vector (KAT2) and the CBC-chain check. This channel is how the expensive KAT2 is exercised: the alternative, driving one million iterations through the differential protocol, would add a million block encryptions per language for a check the repository already performs. We record the self-test wall-clock time as well, because it is the only performance number in the paper and it is reported as a cost, not as a benchmark.

3.6 Mutation study

A null result needs a calibration. We take the harness's own reference model and introduce 294 single-point faults: 256 that flip one bit of one S-box entry, 32 that flip one bit of one round constant, 4 that move one rotation amount in L by one, and 2 that move one rotation amount in L' by one. For each mutant we record whether the standard vector notices, and the index of the first differential case that notices, which gives detection as a function of corpus size for prefixes of length $1, 4, 16, 64, 256$.

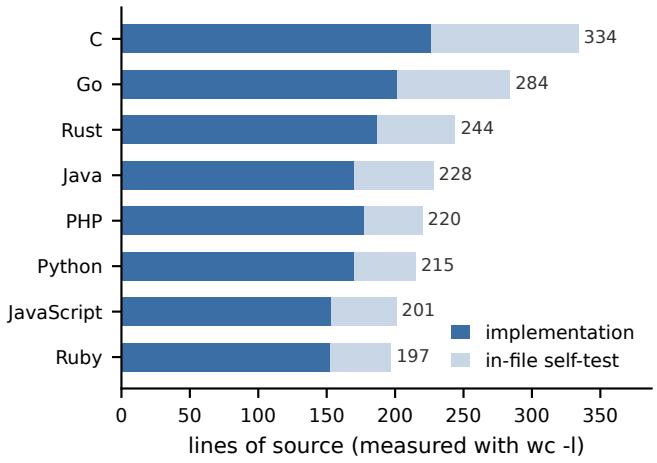


Figure 1: Source lines per implementation, split at the first line of the in-file self-test. For C the implementation bar is `sm4.c` plus `sm4.h` and the self-test bar is `test_main.c`. Measured with `wc -l`; the rule is applied by `scripts/differential.py` and stored in `data/differential.json`. The seven-line `sm4.d.ts` declaration file is not counted.

The mutants are faults in the *model*, not in the shipped code. The study therefore measures the detection power of a corpus, not the defect density of SM4Kit. §6 says so again where it matters.

4 Implementation Notes

All eight implementations follow the same plan: assemble four big-endian words from the block, run 32 rounds, write the words back in reverse order, and reverse the round keys for decryption. What differs is how each language represents a 32-bit word, where the S-box lives, and when the round constants are computed. Table 1 summarises those choices; Figure 1 gives the size of each file.

JavaScript. One UMD-wrapped file that runs under Node and in a browser. The subtlety is that JavaScript's bitwise operators are signed 32-bit operations, so every value that crosses a word boundary is followed by `>>>0`; the S-box is a plain array and the 32 constants are computed once when the module loads. The TypeScript surface is a seven-line `sm4.d.ts` declaration file, not a second implementation — the paper counts JavaScript as one language, not two.

Python. Integers are unbounded, so the implementation masks with `0xFFFFFFFF` after each rotation, each S-box substitution and each word assembly. Buffers are `bytes` and `bytearray`; padding and unpadding are explicit functions that validate every PKCS#7 byte.

Table 1: How each implementation represents the three things that cannot be expressed directly in the standard: a 32-bit word, the S-box, and the round constants. Read from the sources; no value here is inferred.

Language	32-bit word arithmetic	S-box storage	CK_i	byte buffer
JavaScript	`Number`; « is signed, »>`0` after every operation	array literal of 256 numbers	precomputed once at module load	`Uint8Array` / `Buffer`
Python	arbitrary precision; `& 0xFFFFFFFF` after every operation	list of 256 integers	tuple precomputed at import	`bytes` / `bytearray`
Go	`uint32`	`[256]byte`	`[32]uint32` built by an init closure	`[]byte`
Java	`int` (signed); »> for shifts, `& 255` on bytes	`int[256]`	static initialiser block	`byte[]`
Rust	`u32`, `rotate_left`	`[u8; 256]`	computed per round by `fn ck(i)`	`Vec` / slice
PHP	64-bit `int`; masked to 32 bits after every operation	`const` array of 256 integers	computed per round in the schedule	`string`
Ruby	arbitrary precision; `& MASK` after every operation	frozen array of 256 integers	computed per round via a lambda	`String` (binary)
C	`uint32_t`, unsigned throughout	`static const uint8_t[256]`	computed per round by `ck(i)`	`uint8_t*` + length

Go. The cleanest arithmetic of the eight: `uint32` wraps at the right width and shifts need no masking. The file is `package main` and contains both the library and a demonstration `main()`, which means it can be copied into a program but cannot be imported as a Go package without renaming the package. We regard that as a real limitation of the distribution, not a stylistic choice. Round constants are computed once in a package-level initialiser.

Java. Two conversions carry the risk: `byte` is signed, so `word()` masks each byte with `& 255`, and `int` is signed, so rotations use »>. The S-box is an `int[256]`; the constants are filled in a static initialiser; buffers are `byte[]` and the public API is a set of static methods on a final class.

Rust. The only implementation that does not tabulate the round constants: `fn ck(i)` recomputes the four bytes of CK_i on each key-schedule round, which is a few extra multiplications and removes 32 constants from the source. The API returns `Result` for decryption because padding validation can fail, and the four tests live in a `#[cfg(test)]` module.

PHP. PHP's integers are 64-bit, so every shift, XOR and rotation is followed by `& 0xFFFFFFFF`; this is the implementation where forgetting one mask would be least visible, because the language is happy to carry the extra bits. Byte strings are native strings, and CBC's XOR is the language's string XOR operator. Round constants are derived inside the key schedule.

Ruby. Arbitrary-precision integers again, with a `MASK` constant applied after each operation; words are packed with `pack('N4')`, which is also the clearest statement of the big-endian convention in the whole repository. The module is a singleton with `module_function` methods, and the S-box array is frozen.

C. `uint32_t` throughout, so shifts cannot overflow a signed type; the S-box is a file-static `const uint8_t[256]`. This is the only implementation whose header exposes block-level encryption and decryption (`sm4_encrypt_block`, `sm4_decrypt_block`) in addition to the padded API, and the only one whose self-test lives in a separate translation unit (`test_main.c`) rather than in the implementation file.

What is shared. All eight avoid native-endian word loads, all eight validate PKCS#7 padding on decryption, and all eight obtain decryption by reversing the round-key array rather than by inverting the round function. Those three decisions are what make a byte-for-byte comparison meaningful in the first place.

5 Results

5.1 What was executed

Table 2 lists the eight implementations, the toolchain found on the authoring host (Windows 11, x86-64), and what ran. Four languages were executed: JavaScript (Node v24.21.0), Python (CPython 3.13.14), Java (javac and Java 27) and Rust (rustc 1.98.1). Four were not: Go, PHP, Ruby and C have no interpreter or compiler on this machine, so their drivers were shipped but never run.

5.2 The differential result

For the four executed implementations the harness reports zero mismatches in every class:

- 259 block cases per language, each producing a 32-byte ECB ciphertext. That is 518 block encryptions per language and 2,072 across the four.

Table 2: Execution status on the authoring host. "Driver" is the wall-clock time of the differential driver process, including interpreter start-up; Python's driver runs in-process and has no separate time. "Self-test" is the implementation's own suite, run with `-long` where the repository supports it.

Language	Toolchain	Driver	Self-test
JavaScript	Node v24.21.0	0.07 s	PASS, 0.30 s
Python	CPython 3.13.14	in-process	PASS, 48.3 s
Go	not installed	not run	not run
Java	javac 27, Java 27	0.17 s	PASS, 0.39 s
Rust	rustc 1.98.1	0.56 s	4/4 tests, 1.01 s
PHP	not installed	not run	not run
Ruby	not installed	not run	not run
C	no C compiler	not run	not run

- $259 \times 32 = 8{,}288$ round-key words per language (33,152 in total), compared word by word against the reference and against every other language.
- 9 ECB and 9 CBC payload outputs per language, covering 70 block encryptions per language once padding is accounted for.
- All six language pairs agree on every one of the 259 ciphertexts and every one of the 8,288 round-key words; the pairwise comparison performs 49,728 round-key comparisons and finds no difference.

The standard vector is reproduced exactly by all four, and by the reference model. The full result is in `data/differential.json` and `data/differential.csv`; the raw driver transcripts are in `data/transcripts/`, one file per language, so a reader can diff them directly.

Figure 2 shows the matrix as the harness writes it. The grey row segments are the four languages that could not be run; they are visibly different from the green cells because absence of evidence is not evidence of conformance.

The self-test channel agrees: KAT1, KAT2 and the CBC-chain check pass in all four executed languages. The million-iteration vector is the reason Python's self-test takes 48.3 s while Java's takes 0.39 s and Rust's 1.01 s — the same arithmetic, three very different constant factors. We report the times because they were measured, not because they are a benchmark of the languages.

5.3 Static conformance across all eight

Executing four languages leaves four unexamined, so the harness also reads all eight sources. A parser locates the S-box declaration in each file, extracts the 256 byte values, and checks three properties:

- **Size and bijectivity.** All eight tables have exactly 256 entries and all eight are permutations of $0, \ldots, 255$. A single mistyped nibble that duplicates another entry would fail this check without any execution.
- **Cross-source identity.** All eight tables are byte-identical; the SHA-256 of the 256-byte sequence is `1674d0631a2e37b60ec748bc3899fee1550c108af30b85ce96a70c5ca4682d03` for every implementation.
- **Constant presence.** All eight contain the FK constant `a3b1bac6` (Rust with digit separators) and derive CK_i from the $(4i + j) \cdot 7 \bmod 256$ rule.

This is the strongest statement the paper can make about Go, PHP, Ruby and C: their tables are the same table, and it is a bijection. It is not a statement that their round functions compute the right cipher.

Table 3: Mutants detected, by mutant class and corpus size. KAT1 is the single GB/T 32907 Annex A vector; the remaining columns are prefixes of the seeded 256-case differential corpus. All values are counted by `scripts/differential.py` (`data/mutation.json`).

Mutant class	n	KAT1	1	4	16	64	256
S-box entry flip	256	206	211	255	256	256	256
Round constant bit	32	32	32	32	32	32	32
L rotation off by one	4	4	4	4	4	4	4
L' rotation off by one	2	2	2	2	2	2	2
Total	294	244	249	291	294	294	294

5.4 Detection power of the corpus

Table 3 and Figure 3 report the mutation study. The single standard vector detects 244 of the 294 mutants (83.0%). It detects every structural mutant — all 32 round-constant faults, all four L rotation faults and both L' rotation faults — because a wrong rotation or a wrong constant changes every ciphertext. What it misses is table entries: it detects 206 of 256 single-entry S-box faults, so 50 table mutations produce the correct standard ciphertext and are invisible to a vector-only test suite.

The differential corpus closes that gap quickly. One random case detects 211 of 256; four cases detect 255; sixteen cases detect all 256. Across all classes, four cases take detection from 83.0% to 99.0%, and sixteen cases to 100%. Not one of the 294 mutants survives 16 random keys and blocks.

The asymmetry is the point. A vector-only suite is good at catching the faults that change everything and bad at catching the one that changes one entry. A vector-only suite therefore reports "KAT passes" exactly as confidently for a correct implementation and for one with a wrong table entry that the vector happens not to index.

5.5 What the null result does and does not establish

It establishes that four implementations written in four languages, with four different integer models and four different buffer representations, produce identical ciphertexts, identical round keys and identical mode outputs on 259 block cases and 9 payloads, and that all four reproduce the standard. It establishes that the eight S-box tables are the same bijection. It

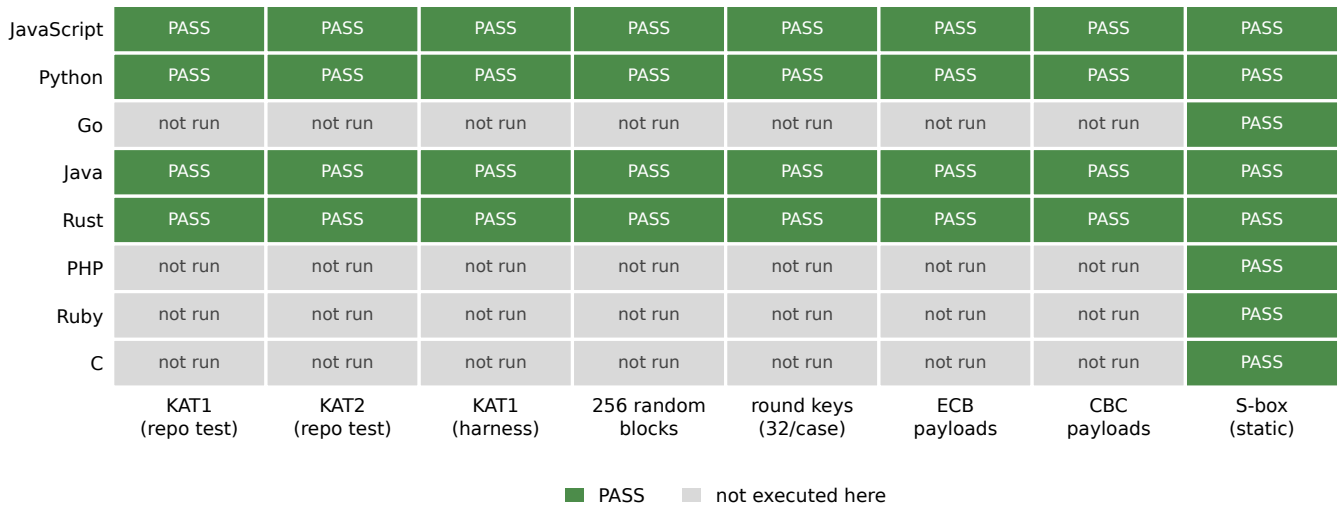


Figure 2: Pass/fail/not-executed matrix, generated from `data/differential.json`. "repo test" is the implementation's own self-test (KAT1 and the million-iteration KAT2); the remaining columns are driven by the differential harness. Go, PHP, Ruby and C were not executed on the authoring machine; their only green cell is the static S-box check, which does not execute the cipher.

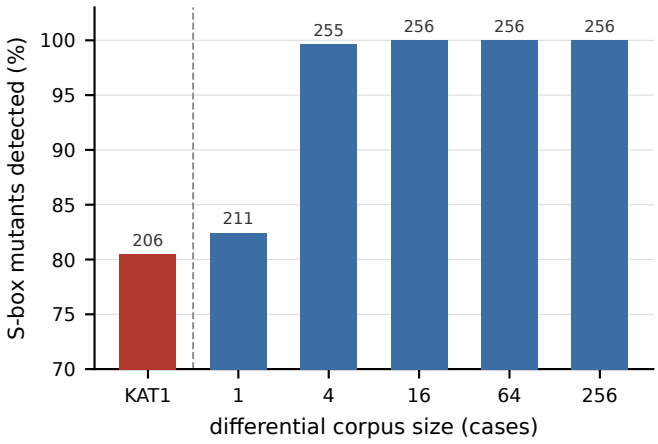


Figure 3: Detection of the 256 single-entry S-box mutants as a function of corpus size. The dashed line separates the single standard vector from the differential corpus. The labels are counts, not percentages: KAT1 detects 206, one random case 211, four cases 255, and sixteen cases all 256. The three structural mutant classes of Table 3 are detected at 100% by every corpus, including KAT1, and are therefore not plotted.

does not establish that the four unexecuted implementations compute the correct cipher, and it does not establish that the four executed ones are free of faults that no test in this corpus reaches — §6 enumerates the ones we know about.

6 Threats to Validity

Half the matrix was not executed. The dominant threat is the obvious one: Go, PHP, Ruby and C were not run. Their drivers exist and are part of the artefact, but a driver that has never been compiled is not evidence. The paper's claims about those four are limited to the static table checks, and every table and figure says so.

Common-mode error is invisible to this method. The eight implementations were written by one author from one standard, so they are separate but not independent in the N-version sense. A misunderstanding shared by all eight would appear as agreement. The reference model does not help: its S-box was transcribed by the same author. This is not a hypothetical corner. The standard vector indexes 206 of 256 S-box entries; if all eight tables shared a typo in one of the remaining 50 entries, the KAT would pass and the differential comparison would find nothing, because there would be nothing to disagree with. The only defence in this study is that the table is a bijection and that 206 entries are pinned by the standard vector. A reader who wants more should compare against an independently transcribed table or an algebraic construction of the S-box, which we did not do.

The drivers are new code. Seven drivers and a reference model were written for this paper. A defect in a driver could mask a divergence: if two drivers made the same formatting mistake, the comparison would pass. The mitigation is that each driver is small, calls only the public API, and its output is committed verbatim as a transcript; a reader can check one line of any transcript by hand.

One host, one toolchain per language. All four executed languages ran under one version on one operating system and one architecture. We did not test a 32-bit platform, a big-endian platform, or a second version of any interpreter. The

harness makes that cheap to repeat, but this paper does not report it.

The corpus is random, not adversarial. 256 random keys and blocks are a sample, not a search. A fault that only manifests on a structured input — a low-weight key difference, a block with a repeated byte pattern — could survive. The all-zero and all-`0xFF` cases are the only structured inputs besides the standard vector.

Only the public surface is compared. The round registers X_i are private in all eight implementations. A divergence that cancels before the output, or one confined to a round that the output does not distinguish, would not be seen. The round-key comparison partially covers the schedule; nothing covers the round function's internals.

The self-test channel is not audited. We record what each repository's self-test reports; we did not evaluate whether those assertions are strong. JavaScript and Python, for instance, run the boundary round-trips but do not report them individually, which is why the corresponding CSV cells read `N/A` rather than PASS.

The mutation study mutates the model. It measures the detection power of a corpus against single-point faults in one implementation of the specification. It says nothing about how often such faults occur in practice, and its 83.0% figure is a property of this vector and this mutant set, not a general law.

7 Limitations

The limitations below are specific and, where the repository already states them, we repeat the repository's own wording rather than softening it.

- **Four languages unexecuted.** Go, PHP, Ruby and C have drivers but no results. The matrix reports them as not executed. A reader with those toolchains can complete the study by running the harness; nothing else needs to change.
- **No TypeScript implementation.** The TypeScript surface is a seven-line declaration file for the JavaScript module. There is no second TypeScript source tree to compare, and the paper counts one JavaScript implementation, not two.
- **Confidentiality only.** The library implements ECB and CBC with PKCS#7. It has no MAC, no AEAD, no CTR or GCM mode, and no key derivation. CBC with PKCS#7 and no MAC is exposed to padding-oracle attacks; the repository's security notes say so, and the harness does not test error paths, so the paper has nothing to add about them.
- **Not constant-time.** The S-box is indexed with secret-dependent data in all eight implementations. Cache-timing and other microarchitectural channels are out of scope and were not measured Kocher (1996).
- **Go is not importable.** `go/sm4.go` is `package main`, so it is a copy-in file rather than an importable package. This is a distribution limitation, not a correctness one, and the paper does not count it as a feature.
- **Error paths are not compared.** The eight implementations report errors differently — exceptions, `Result`, an error out-parameter, returned `None` — and the messages are in Chinese. The harness drives only successful encryptions, so cross-language error behaviour is untested.
- **KAT2 is driven by the self-tests.** The million-iteration vector is exercised through each repository's own suite, not through the differential harness. That keeps the harness cheap but means KAT2 is attested by the implementation under test rather than by an external driver.
- **No CI evidence.** The repository has a GitHub Actions workflow for all eight languages, but this paper reports only what was run locally. We did not use the workflow as evidence, because we did not observe it run.
- **One author, one machine.** Every measurement comes from a single Windows 11 host. No claim is made about reproducibility on other platforms beyond the fact that the harness is seeded and the corpus is committed.

8 Related Work

Differential testing as a technique is due to McKeeman McKeeman (1998); Csmith Yang et al. (2011) is its best-known large-scale application, and pivoted query synthesis Rigger and Su (2020) is a recent cryptographic-adjacent example in database engines. The oracle problem — what to compare against when no implementation is trusted — is surveyed by Barr et al. Barr et al. (2015). This paper uses three oracles at once: the standard vector, an independent model, and the implementations against each other.

N-version programming Avižienis (1985) also runs several versions of one specification, but to survive a fault rather than to expose one. Our observation that separate implementations written by one author are not independent is the standard critique of N-version programming applied to a much smaller artefact.

Randomised testing in the QuickCheck style Claessen and Hughes (2000) and the fuzzing literature Manès et al. (2019) supply the corpus-generation discipline; the corpus here is deliberately small and seeded, because the point is reproducibility rather than coverage.

On the cryptographic side, CAVP/ACVP National Institute of Standards and Technology validates implementations against known-answer and property tests, Wycheproof Google curates edge cases that break real libraries, and Cryptofuzz Vranken runs libraries against each other. All three operate within a language and a build system. The axis varied here is the language itself, which is where the signedness, width and buffer-convention faults of §2.3 live. We are not aware of a published cross-language differential study of SM4, and we make no claim that none exists.

SM4 itself is specified in GB/T 32907-2016 Standardization Administration of the People's Republic of China (2016); AES plays the same role in the Western literature and has an analogous known-answer culture National Institute of Standards and Technology (2001). The byte-order conventions that make cross-language comparison non-trivial are the subject of Cohen's classic note Cohen (1981).

9 Conclusion

Eight implementations of one standard are eight chances to disagree, and a single test vector is a poor instrument for finding out. This paper built the instrument: a differential harness that drives every implementation through the standard vector, a seeded random corpus and boundary-length payloads, and that compares ciphertexts, the full round-key schedule and mode outputs against a reference model and against each other.

The measured result is a null result, and we have kept it that way. The four implementations we could execute — JavaScript, Python, Java and Rust — agree exactly on 259 block cases, 8,288 round keys and 18 mode outputs per language, with no difference in any of the six pairs. The static check finds one 256-entry bijective S-box table, identical in all eight sources. The four implementations we could not execute — Go, PHP, Ruby and C — are reported as not executed, in the paper and in the machine-readable results, because that is what they are.

The calibration is what turns the null result into an argument. The standard vector detects 244 of 294 single-point mutants and misses 50 of the 256 S-box table faults; sixteen random cases detect all 294. A vector-only test suite is therefore not merely weaker than a differential one — it is blind in a specific, measurable place, the part of the table that the vector never indexes. The harness, the corpus, the drivers, the transcripts and the scripts that produced every number here are part of the submission, so that the four missing toolchains can be filled in and the blind spot can be probed from a second direction.

References

A. Avižienis. The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11 (12):1491–1501, 1985.

E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5): 507–525, 2015.

K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 268–279, 2000.

D. Cohen. On holy wars and a plea for peace. *IEEE Computer*, 14(10):48–54, 1981.

M. Dworkin. Recommendation for block cipher modes of operation: Methods and techniques. NIST Special Publication 800-38A, 2001.

Google. Wycheproof: Test vectors for cryptographic libraries. https://github.com/google/wycheproof. Accessed October 2026.

B. S. Kaliski. PKCS #7: Cryptographic message syntax version 1.5. RFC 2315, 1998. IETF.

P. C. Kocher. Timing attacks on implementations of Diffie–Hellman, RSA, DSS, and other systems. In *Advances in Cryptology — CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

V. J. M. Manès, H. Han, C. Han, et al. The art, science, and engineering of fuzzing: A survey, 2019. arXiv:1812.00140.

W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1), 1998.

National Institute of Standards and Technology. Automated cryptographic validation protocol (ACVP). https://github.com/usnistgov/ACVP. Accessed October 2026.

National Institute of Standards and Technology. Advanced encryption standard (AES). FIPS Publication 197, 2001.

M. Rigger and Z. Su. Testing database engines via pivoted query synthesis. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

Standardization Administration of the People's Republic of China. GB/T 32907-2016: Information security technology — SM4 block cipher algorithm, 2016. National standard of the People's Republic of China.

G. Vranken. Cryptofuzz: Differential fuzzing of cryptographic libraries. https://github.com/guidovranken/cryptofuzz. Accessed October 2026.

X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 283–294, 2011.