Typhon DB Sentry: Practical Queryable Column-Level Encryption with Fast Format-Preserving Decryption

Roberto Gallo[1]

[1]KRYPTUS Information Security SA

gallo@kryptus.com

Abstract. *Querying data that stays encrypted is fiercely difficult—cryptographic, semantic, and performance demands collide. Column-level encryption shields sensitive fields from privileged insiders but destroys the semantic properties SQL relies on, and—when made queryable—proxy-side aggregation becomes the bottleneck, as the proxy must decrypt every value before folding it. We present* ***Typhon DB Sentry****, a transparent wire-protocol SQL proxy that makes column-encrypted data queryable (range, pattern matching, ordering, aggregation) across PostgreSQL, MySQL, SQL Server, Oracle and IBM Db2, with keys kept outside the application. Its core tool contribution is a* fast format-preserving decryption *path: we replace the arbitrary-precision (big-integer) radix arithmetic of NIST FF1 with an* adaptive fixed-width *implementation—native 128-bit integers on the common case, big-integer fallback otherwise—which, to our knowledge, is the first* adaptive *radix arithmetic for FF1,* byte-identical *to the standard; and we decrypt large result sets* in parallel*, fusing decryption and aggregation into a single pass. With the full path, proxy-side* `SUM`/`AVG` *over 150K encrypted values runs in ∼0.08–0.12 s across the five dialects (∼1.6–18× over cleartext; conventional decrypt-then-fold pays ∼80×), without changing ciphertext. We demonstrate the tool with a complete medical clinic web application served entirely through the proxy.*

Resumo. *Consultar dados mantendo-os cifrados é ferozmente difícil—exigências criptográficas, semânticas e de desempenho colidem. A cifragem em nível de coluna protege dados sensíveis contra adversários privilegiados, mas destrói as propriedades semânticas do SQL; quando se torna consultável, a agregação no proxy vira o gargalo, pois é preciso decifrar cada valor antes de agregá-lo. Apresentamos o* ***Typhon DB Sentry****, um proxy SQL transparente que torna dados cifrados por coluna consultáveis (faixa, padrão, ordenação, agregação) em PostgreSQL, MySQL, SQL Server, Oracle e IBM Db2, com as chaves fora da aplicação. A contribuição central da ferramenta é um caminho de* decifragem rápida com preservação de formato*: trocamos a aritmética de precisão arbitrária do FF1 (NIST) por uma implementação de* largura fixa adaptativa*—inteiros de 128 bits no caso comum, com* fallback *para inteiros grandes—*byte-idêntica *ao padrão, e decifra mos grandes conjuntos de resultados* em paralelo*, fundindo decifragem e agregação em um único passo. Com o caminho completo, o* `SUM`/`AVG` *no proxy sobre 150K valores cifrados executa em ∼0,08–0,12 s nos cinco dialetos (∼1,6–18× sobre o texto claro;*

o caminho convencional decifra-e-agrega paga ∼80×), sem alterar o texto cifrado. Demonstramos a ferramenta com uma aplicação web completa de clínica médica servida inteiramente através do proxy.

1. Problem and Motivation

Column-level encryption gives fine-grained protection to sensitive fields and, unlike full-disk encryption, defends against the adversary that matters most—the *privileged insider*: a database administrator with full storage access sees only ciphertext. This is central to data-protection regimes such as the LGPD and the GDPR. The catch is unforgiving: the very properties that make a relational database useful—order, substring containment, and equality—vanish the moment plaintext becomes ciphertext, and `WHERE`, `LIKE`, `ORDER BY` and aggregates stop working.

Schemes that keep queries working over encrypted data exist [4, 6, 1], but a specific cost has been treated as inherent: *aggregation*. Because a transparent proxy cannot sum ciphertext, `SUM`/`AVG` must decrypt the whole column proxy-side before folding—an $O(N)$ decryption whose constant, with format-preserving encryption (FPE-FF1), is dominated by big-integer radix arithmetic. On a ∼1M-row workload this is ∼1.4 s, ∼80× slower than cleartext, and is commonly taken as a practical ceiling on proxy-side aggregation.

This paper presents **Typhon DB Sentry**, a production-grade, transparent SQL proxy, and shows that this ceiling is not fundamental: it is a *reducible constant*. Our tool contribution is a fast FPE decryption path—adaptive fixed-width radix arithmetic, parallel bulk decryption and a fused decrypt-and-fold pass—that is byte-identical to NIST FF1 and brings the same aggregation to ∼0.08–0.12 s (∼1.6–18×) inside a working multi-dialect proxy. A companion paper [1] introduces the blind-index scheme and its leakage bounds; in that work, proxy-side aggregation is the heaviest case, carried at full $O(N)$ cost. **This paper's advance is to reduce that constant**—the before/after is the ablation in Table 1, with output byte-identical to NIST FF1. Here we focus on the tool and this decryption contribution. As a Salão de Ferramentas submission, the primary artifact is a working, demonstrable tool; as added value, it contributes a fast, standard-compatible FF1 decryption technique not, to our knowledge, previously documented for that mode.

2. Typhon DB Sentry: Overview

Typhon DB Sentry is a wire-protocol SQL proxy interposed between the application and the database (Figure 1). The application keeps its native driver and unmodified SQL; the operator points the connection string at the proxy. For each column declared in a TOML policy, the proxy encrypts data on writes, rewrites predicates on reads so the database pre-selects rows using its native indexes, and decrypts and verifies results before returning plaintext. Keys and role-based access control live *in the proxy*, outside the application's trust domain—a compromised application cannot exfiltrate keys or bulk-decrypt the dataset.[1] Protection types are `aes_gcm`, `fpe_ff1` (format-preserving, NIST SP 800-38G [9]), `tokenize` and `mask`. The proxy is ∼47K lines of memory-safe Rust on the

[1]This holds cleanly for the wire-protocol backends (PostgreSQL, MySQL, SQL Server); the Oracle and Db2 paths use a sidecar invoked by a wrapper driver in the application environment and therefore retain a client-side trust profile.

Tokio runtime; a shared, dialect-independent rewrite core (via `sqlparser-rs` [16]) serves PostgreSQL, MySQL/MariaDB, SQL Server (a from-scratch TDS implementation), and Oracle and IBM Db2 LUW (each through the same sidecar architecture with a thin wrapper driver).

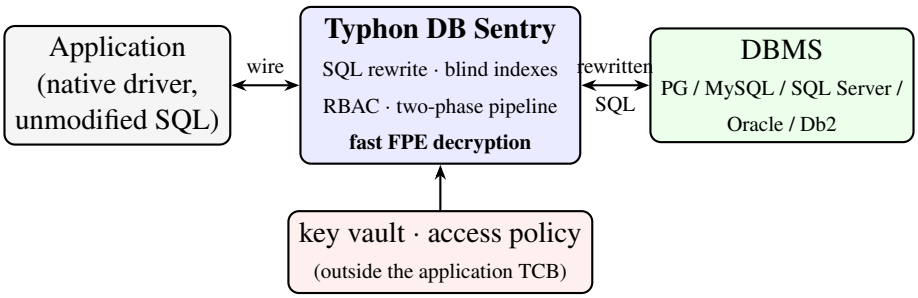

Figure 1. Typhon DB Sentry interposes at the wire-protocol level: the application issues ordinary SQL; keys and access control stay outside it.

Queryability in one paragraph. Two auxiliary columns make encrypted data queryable. An *order-hint* bucket—built with minimum-occupancy bucketization, in the style of De Capitani di Vimercati et al.'s flat indexes [3], with bucket identifiers permuted under a key—lets the proxy recover approximate order for range and `ORDER BY` while the server sees neither plaintext order nor bucket rank; a keyed *blind substring* index enables `LIKE`/`ILIKE`. A two-phase pipeline guarantees correctness: Phase 1 rewrites predicates over the auxiliary columns and *widens* operators so the database returns a superset of the answer (zero false negatives); Phase 2 decrypts the qualifying rows, verifies the original predicate exactly, and—for `SUM`/`AVG`/`COUNT`—folds the aggregate *after decryption*. Phase 2 is where decryption cost concentrates, and it is the subject of Section 4.

3. The Format-Preserving Decryption Bottleneck

FF1 encrypts an n-numeral string in base *radix* with a ten-round Feistel network whose round function is an AES-CBC-MAC PRF [9, 10]. The data-dependent cost is not the AES—which is cheap with AES-NI—but the conversion between numerals and integers at every round: NUM_{radix} (Horner accumulation), STR_{radix} (repeated division), and the modular step $(\mathrm{NUM}(A) \pm y) \bmod radix^m$. Reference implementations (and the Rust `fpe` crate [14] we build on) perform this on *arbitrary-precision big integers*, which allocate a heap vector per operation and run general-purpose digit algorithms even when the value is tiny.

Profiling the proxy under a decryption-heavy aggregate confirms it: with FPE columns, the radix conversion and its allocations dominate proxy CPU, while AES-NI is a few percent. Yet real FPE domains are *small*—an 11-digit national identifier is $\sim 10^{11} < 2^{37}$, well within a 64/128-bit word. Arbitrary precision is correctness insurance paid on every value, and that is the constant we attack.

4. Fast Format-Preserving Decryption

Two techniques in the shared crypto layer—active for all five dialects—plus a fused decrypt-and-fold in Phase 2 turn proxy-side aggregation from a limit into a tunable cost. Figure 2 shows the Phase-2 path they accelerate.

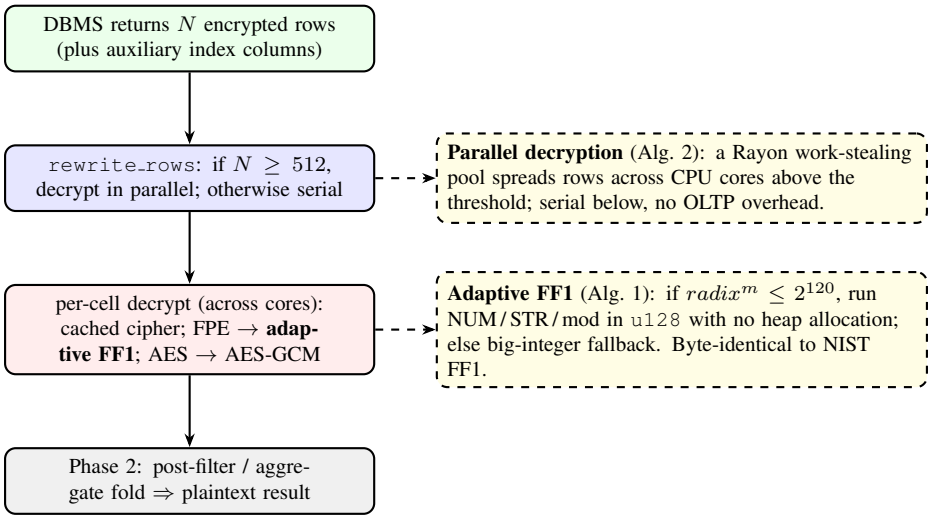

Figure 2. Phase-2 decryption path. Large result sets fan out across cores (`rewrite_rows`); each cell uses a prebuilt cipher, and FPE cells run the adaptive fixed-width FF1 arithmetic with a big-integer fallback.

4.1. Adaptive fixed-width radix arithmetic

We do *not* reimplement FF1: the Feistel structure and the AES-CBC-MAC PRF are the upstream library's, unchanged. The `fpe` crate exposes the numeral string through a generic interface, so we provide a drop-in implementation that does the radix arithmetic in machine words whenever it is overflow-safe and delegates to the original big-integer code otherwise (Algorithm 1). Because both paths compute the *same integers*, ciphertext is unchanged—existing data stays decryptable. This matters practically: published FPE libraries apply this fast path, if at all, only to FF3 [13]—the mode our scheme deliberately avoids, FF3 having been weakened by small-domain attacks [11]—whereas for FF1 this *adaptive* design (fast path with big-integer fallback) is, to the best of our knowledge, a first.

The fallback is not a corner case to fear but a correctness guarantee: any domain—a long account string, a high radix—that does not fit a word runs the exact original code. Byte-identity across radixes, lengths spanning the fast-path/fallback boundary, and the 0/$radix-1$ extremes is asserted by an exhaustive cross-check test against the upstream numeral string.

4.2. Parallel bulk decryption

Decryption of a result set is embarrassingly parallel: each row is independent. The shared helper `rewrite_rows` (Algorithm 2) fans rows out across CPU cores (via the Rayon work-stealing pool [15]) above a threshold and stays serial below it, so latency-sensitive OLTP queries pay no thread-pool overhead while bulk aggregation scales with cores. Safety under parallelism is structural: the crypto engine's hot path is read-only (prebuilt per-key ciphers, read-locked caches, per-worker scratch), so no synchronization is needed in the inner loop. All five backends route their bulk decryption through this one helper.

Algorithm 1: Adaptive numeral arithmetic, plugged into upstream FF1.

require: base $radix$; half-length m; bound $\tau = 2^{120}$ (keeps $acc \cdot 256$ in `u128`)

function NUM(X): ; `// numerals → integer`
  $acc \leftarrow 0$; **for** $x \in X$ **do**
    $acc \leftarrow acc \cdot radix + x$ **with overflow check**; **on overflow return** BigNUM(X)
  return acc ; `// a u128 word`

function STR(c, m): ; `// integer → m numerals`
  if *c is a word* **then for** $i \leftarrow m$ **to** 1 **do**
    $out_i \leftarrow c \bmod radix$; $c \leftarrow c \div radix$
  ;
  else return BigSTR(c, m);
return out

function ADDMOD(A, y_{bytes}, m): ; `//` $(\text{NUM}(A) + y) \bmod radix^m$
  $M \leftarrow radix^m$ **(checked)**; **if** $M \leq \tau$ ***and*** NUM(A) *is a word* **then**
    $a \leftarrow \text{NUM}(A)$; $y \leftarrow \text{reduce\_be}(y_{\text{bytes}}) \bmod M$;
    return STR$\big((a + y) \bmod M,\ m\big)$
  return BigADDMOD(A, y_{bytes}, m) ; `// byte-identical fallback`

Algorithm 2: `rewrite_rows`: adaptive parallel decryption (shared by all backends).

Input: rows R; column policies; crypto engine E (prebuilt ciphers, read-only)
if $|R| \geq$ THRESHOLD (512) **then**
  parallel for $r \in R$; `// Rayon work-stealing over cores`
    decrypt_row(r, E) ; `// per cell: cached cipher; FPE via Alg. 1`
else
  for $r \in R$: decrypt_row(r, E) ; `// serial: OLTP, no overhead`

5. Demonstration

We demonstrate the tool with a **complete medical-clinic web application** served entirely through the proxy, packaged as a Docker Compose stack (database, key vault, onboarding, proxy, and the app). The application is plain Go (`net/http` + `lib/pq`, an HTMX/Material-Design front end) over PostgreSQL: ∼5,000 synthetic patients, 30 doctors, 25K appointments, billing and clinical records, with 18 encrypted columns (national identifiers and phones as FPE-FF1; names, e-mails, clinical notes and billing amounts as AES-GCM). No real personal data is used. **The application imports no cryptography and does not know which columns are encrypted**—it just speaks SQL; every protection decision is the proxy's. A technical video covering installation, operation and results is available at `https://youtu.be/ePS9BfeoA_0`.

The live walk-through has four acts:

1. **Role-based views of one row.** Open a patient and switch the header role among *doctor*, *receptionist* and *DBA*. The *same* query returns plaintext, an RBAC-restricted value (clinical notes become "restricted"), or raw ciphertext—enforced at the proxy, with no change in the application (Figure 3).
2. **Encrypted search.** Search patients by name: a `LIKE` over the AES-GCM `full_name` resolved through the keyed blind substring index (Figure 4); and

(a) As *doctor*—the proxy decrypts the matching row.

(b) As *DBA*—identical query and rewrite, ciphertext result.

Figure 3. The interactive SQL console (Acts 1 and 4): an arbitrary query runs through the proxy and its rewrite is shown verbatim. Equality on the FPE-FF1 `cpf` is rewritten by *encrypting the search term in place* (`56956160127`→`38000840773`, the 11-digit format preserved), so the predicate matches on ciphertext. The same query yields plaintext for a clinician (a) but format-preserving and AES-GCM ciphertext for the DBA (b)—RBAC and encryption enforced wholly at the proxy. Tellingly, the DBA's result `cpf` *equals* the rewritten predicate: the match really did run on ciphertext.

by national identifier via FPE equality. Each query exercises fast FPE decryption on the qualifying set.

3. **Aggregation over encrypted data—the contribution, live.** The billing dashboard runs `SELECT specialty, COUNT(*), SUM(amount), AVG(amount) GROUP BY specialty ORDER BY total DESC`. The proxy decrypts every `amount` and folds the aggregate in memory (the accelerated Phase-2 path of Section 4); the doctor sees real per-specialty revenue. As a DBA, the *same* query returns "— encrypted —": the proxy honestly cannot fold values it may not decrypt for that role.
4. **Show the rewrite.** A "Show SQL" panel renders *Original* → *Materialized* → *Rewritten* side by side, exposing how `GROUP BY`, `ORDER BY` and `LIMIT` are lifted into a proxy-side fold and how `LIKE` is redirected to the blind-index column.

Equipment. A laptop running Docker Compose and a projector; no network is required.

6. Evaluation

We report measurements on a separate 1M-row synthetic medical workload (seven tables, 14 encrypted columns; larger than the demo of Section 5), all five backends co-located on a commodity workstation (AMD Ryzen AI Max+ 395, 32 threads; Table 1's ablation: Intel Core i9, 24 threads); the companion paper [1] has the full 23-query benchmark.

Figure 4. Encrypted substring search (Act 2). A `LIKE '%Harris%'` over the AES-GCM `full_name`—which no deterministic or order-preserving scheme can answer—is rewritten to a keyed blind-index membership pre-filter over per-n-gram buckets; the proxy then decrypts, verifies the predicate exactly, and returns only the qualifying rows.

We isolate the heaviest case—proxy-side SUM/AVG over 150K FPE-encrypted values—and ablate the first two techniques (Table 1). Adaptive fixed-width arithmetic alone cuts latency $\sim$57% ($2.3\times$); adding parallel decryption reaches $\sim$79% ($4.7\times$), taking the overhead from $\sim 80\times$ to $\sim 16\times$—without changing ciphertext.

Table 1. Ablation of the fast-decryption path: proxy-side SUM/AVG over 150K FPE-encrypted values ($\sim$1M-row workload, MariaDB). The baseline row is the unoptimized cost carried by the companion paper [1]; this paper's contribution is the $4.7\times$ reduction below it; the fused pass of Table 2 lowers it further, to $\sim$0.1 s.

Configuration	Latency	Latency cut	Overhead vs. cleartext
Baseline (big-integer FF1, serial)	$\sim$1.4 s	—	$\sim 80\times$
+ adaptive fixed-width FF1	$\sim$0.6 s	$\sim -57\%$	$\sim 35\times$
+ parallel bulk decryption	$\sim$0.3 s	$\sim -80\%$	$\sim 16\times$

The effect is cross-dialect, since the optimization lives in the shared crypto layer. On MariaDB the first two techniques take SUM/AVG from $\sim$1.4 s to $\sim$0.3 s ($\sim 16\times$); a *fused decrypt-and-fold* pass—folding each value into the running aggregate as it is decrypted—then brings it to $\sim$0.1 s, the $\sim$1.6–$18\times$ of Table 2, which reports proxy-side SUM/AVG across all five backends (incl. IBM Db2 LUW) with the full path enabled.

Table 2. Proxy-side SUM/AVG over 150K encrypted values across five dialects, full fast-decryption path ($\sim$1M-row workload). Latency lands at $\sim$0.08–0.12 s on every dialect; overhead over cleartext spans $\sim$1.6–18$\times$, dominated by backend fetch plus $O(N)$ proxy-side decryption.

	PostgreSQL	MySQL	SQL Server	Oracle	Db2
Cleartext (ms)	12.8	21.7	56.5	6.6	17.7
Proxied, fast (ms)	81.2	96.0	90.5	116.7	121.4

Other query classes are unaffected by, or benefit from, the same path: selective queries stay sub-millisecond, multi-table joins stay under $2\times$, and LIKE performs comparably to cleartext (the proxy verifies in memory instead of scanning).

Beyond the synthetic medical workload used for the controlled measurements above, the proxy has been exercised on *real transaction data* and placed, unmodified, in

front of a real-world government case-management application (SEI, a widely deployed PHP/MySQL system), confirming wire-protocol transparency on production-shaped applications and data without any change to the application.

7. Positioning and Comparison

Every approach to querying encrypted data negotiates a three-way trade-off among *confidentiality* against the server, SQL *semantic richness*, and *performance*; no tool maximizes all three. Native and disk-level TDE keep full SQL at native speed but concede confidentiality to the DBA—the very insider that column-level encryption targets. Deterministic FPE and tokenization (Thales CipherTrust, Voltage SecureData) defend against the DBA and preserve equality, but range, `LIKE` and arithmetic break. SMPC proxies (Baffle, Arx [6]) preserve semantics and reduce leakage, yet require dedicated compute parties and pay 5–$100\times$ for inter-party round-trips. Property-preserving databases in the database-service-provider line [2]—CryptDB [4], MONOMI [5]—recovered more semantics at modest reported cost ($\sim$26% on TPC-C; $1.24\times$ median on TPC-H), but inherit order-preserving-encryption leakage [7], which Naveed et al. [8] exploited to recover most plaintexts on medical data. Table 3 places Typhon DB Sentry against representative tools.

Table 3. Typhon DB Sentry among representative SQL-over-encrypted-data tools. ✓ native / fully preserved; (✓) supported with caveats or overhead; ✗ not supported or semantics broken. Overheads are typical orders of magnitude over plaintext.

Capability	Native TDE	Thales C.Trust	Voltage SecData	Baffle (SMPC)	Typhon DB Sentry
Defends against privileged DBA	✗	(✓)	✓	✓	✓
Zero application code	✓	✓	(✓)	✓	✓
Equality (=, `IN`)	✓	✓	✓	✓	✓
Range (<, `BETWEEN`)	✓	✗	(✓)	✓	✓
`LIKE` / `ILIKE`	✓	✗	✗	✓	✓
`SUM` / `AVG` over cipher	✓	✗	✗	✓	✓
No extra compute servers	✓	(✓)	(✓)	✗	✓
Overhead (range / aggregation)	$0\%^{\dagger}$	—	—	5–$100\times$	0.2–$18\times$

[†]Native TDE preserves all SQL at zero overhead precisely because the engine sees plaintext—so it does not defend against the DBA. "—" marks operators that do not run over encrypted data at all.

Typhon DB Sentry occupies a distinct point: a single transparent proxy—no extra compute parties—preserving range, `LIKE` and aggregation across five dialects with bounded, quantified leakage [1], at 0.2–$18\times$ overhead (below $1\times$ for predicates the proxy verifies in memory rather than scanning, up to $18\times$ for full-column aggregation). Aggregation was its one historically weak cell; the fast-decryption path of Section 4 brings proxy-side `SUM`/`AVG` to $\sim$1.6–$18\times$ overhead—at or below the SMPC alternative's 5–$100\times$, without its servers. Among FPE-performance efforts, FF1+ [12] reduces Feistel rounds for constrained devices and fixed-width radix arithmetic has been applied to FF3 [13]; our contribution is to bring it to FF1 *adaptively*—a fixed-width fast path with big-integer fallback, byte-identical to the standard.

8. Final Remarks and Availability

Typhon DB Sentry is a production-grade system—∼47K lines of Rust, five database dialects, transparent to applications—demonstrated by a full medical web application. Its tool contribution is a fast format-preserving decryption path (adaptive fixed-width FF1 with big-integer fallback, parallel bulk decryption, and a fused decrypt-and-fold), byte-identical to NIST FF1, that turns proxy-side aggregation—long taken as a hard ceiling—into a reducible, tunable cost. We are honest about the boundaries: the proxy is part of the trusted computing base; aggregation cost is linear in the number of decrypted values (reduced, not eliminated); minimum-occupancy bucketization bounds order-hint leakage even on skewed columns, though very low-cardinality domains gain little from one; and the scheme accepts bounded, quantified leakage in exchange for queryability. The tool is submitted in the closed-source modality: it is demonstrated live at the venue, with a technical video (`https://youtu.be/ePS9BfeoA_0`) covering installation, operation and results.

References

[1] Gallo, R. (2026). Dual Blind Indexes for Queryable Column-Level Encryption: A Two-Phase Pipeline with Quantifiable Leakage Bounds. In: *Proc. 40th IFIP WG 11.3 Conf. on Data and Applications Security and Privacy (DBSec)*. To appear.

[2] Hacigümüs, H., Iyer, B., Li, C., Mehrotra, S. (2002). Executing SQL over encrypted data in the database-service-provider model. In: *Proc. ACM SIGMOD*, pp. 216–227.

[3] De Capitani di Vimercati, S., Facchinetti, D., Foresti, S., Oldani, G., Paraboschi, S., Rossi, M., Samarati, P. (2024). Multi-dimensional flat indexing for encrypted data. *IEEE Trans. Cloud Comput.*, 12(3), pp. 928–941.

[4] Popa, R.A., Redfield, C.M.S., Zeldovich, N., Balakrishnan, H. (2011). CryptDB: Protecting confidentiality with encrypted query processing. In: *Proc. 23rd ACM SOSP*, pp. 85–100.

[5] Tu, S., Kaashoek, M.F., Madden, S., Zeldovich, N. (2013). Processing analytical queries over encrypted data. *Proc. VLDB Endowment*, 6(5), pp. 289–300.

[6] Poddar, R., Boelter, T., Popa, R.A. (2019). Arx: An encrypted database using semantically secure encryption. *Proc. VLDB Endowment*, 12(11), pp. 1664–1678.

[7] Boldyreva, A., Chenette, N., Lee, Y., O'Neill, A. (2009). Order-preserving symmetric encryption. In: *Advances in Cryptology – EUROCRYPT*, LNCS 5479, pp. 224–241.

[8] Naveed, M., Kamara, S., Wright, C.V. (2015). Inference attacks on property-preserving encrypted databases. In: *Proc. 22nd ACM CCS*, pp. 644–655.

[9] Dworkin, M. (2016). Recommendation for block cipher modes of operation: methods for format-preserving encryption. NIST Special Publication 800-38G.

[10] Bellare, M., Rogaway, P., Spies, T. (2010). The FFX mode of operation for format-preserving encryption. Submission to NIST.

[11] Durak, F.B., Vaudenay, S. (2017). Breaking the FF3 format-preserving encryption standard over small domains. In: *Advances in Cryptology – CRYPTO*, LNCS 10402, pp. 679–707.

[12] Baccarini, A.N., Hayajneh, T. (2019). Evolution of format preserving encryption on IoT devices: FF1+. In: *Proc. 52nd Hawaii Int. Conf. on System Sciences (HICSS)*.

[13] Mysto Project (2021). `ff3`: Format-preserving encryption with native-integer arithmetic. `https://github.com/mysto/python-fpe`.

[14] str4d (2023). `fpe`: Format-preserving encryption (FF1) in Rust. `https://crates.io/crates/fpe`.

[15] Matsakis, N., Stone, J. et al. Rayon: a data-parallelism library for Rust. `https://github.com/rayon-rs/rayon`.

[16] Apache DataFusion. `sqlparser-rs`: an extensible SQL parser for Rust. `https://github.com/apache/datafusion-sqlparser-rs`.