Encrypted Databases Made Secure Yet Maintainable

Mingyu Li, *Shanghai Jiao Tong University; Shanghai AI Laboratory; Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China;* Xuyang Zhao and Le Chen, *Shanghai Jiao Tong University; Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China;* Cheng Tan, *Northeastern University;* Huorong Li and Sheng Wang, *Alibaba Group;* Zeyu Mi, *Shanghai Jiao Tong University; Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China;* Yubin Xia, *Shanghai Jiao Tong University; Shanghai AI Laboratory; Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China;* Feifei Li, *Alibaba Group;* Haibo Chen, *Shanghai Jiao Tong University; Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China*

https://www.usenix.org/conference/osdi23/presentation/li-mingyu

This paper is included in the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-34-2

Open access to the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

Encrypted Databases Made Secure Yet Maintainable

Mingyu Li[1,2,3] Xuyang Zhao[1,3] Le Chen[1,3] Cheng Tan[†] Huorong Li[⋆] Sheng Wang[⋆]
Zeyu Mi[1,3] Yubin Xia[1,2,3] Feifei Li[⋆] Haibo Chen[1,3]
[1]*Shanghai Jiao Tong University* [2]*Shanghai AI Laboratory* [†]*Northeastern University* [⋆]*Alibaba Group*
[3]*Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China*

Abstract

State-of-the-art encrypted databases (EDBs) can be divided into two types: one that protects the whole DBMS engine in a trusted domain, and one that protects only operators that support queries over encrypted data. Both types have limitations when dealing with malicious database administrators (DBAs). The first type either exposes the data to DBAs or makes maintenance operations difficult if the DBA role is eliminated. The second type is vulnerable to abuse of the operator interfaces; in particular, we devise a smuggle attack that enables DBAs to secretly and effectively access data.

We introduce HEDB, which prevents smuggle attacks and preserves database maintainability. HEDB uses a dual-mode EDB design based on our analysis of DBA maintenance tasks. Execution Mode handles user queries by isolating DBAs from operators to prevent smuggle attacks, while Maintenance Mode enables DBMS maintenance and operator troubleshooting through *authenticated replay* and *anonymized replay*, respectively. Our evaluation shows that HEDB blocks smuggle attacks and supports common maintenance tasks with 5.88% runtime cost and 9.26% storage cost.

1 Introduction

With approximately 60 ZB of data stored in database systems [6], much of which is sensitive, data breaches pose one of the most serious security threats today, causing an average loss of $4.35 million per incident [4]. To protect against external attacks, database security features such as role-based access control and encryption at rest have become *de facto* standards. However, these features are not effective at preventing attacks from malicious insiders, who create new internal threats. This is especially true for Database as a Service (DBaaS) scenarios, where cloud platform operators and database administrators (DBA) have full access to the database engine and customer data. To address this threat, several encrypted database (EDB) systems have been proposed by academia [15, 17, 26, 42, 44, 48] and industry [14, 30, 50].

Despite a broad spectrum of prior efforts [14, 15, 17, 26, 30, 42, 44, 48, 50], EDB systems with (i) full-SQL functionality, (ii) maintainability and (iii) strong security have remained an unsolved problem for the past decade. State-of-the-art EDB systems can be largely categorized into two types: (1) a monolithic EDB design that isolates the whole database engine in a trusted domain, and (2) a plug-and-play EDB design that leverages protected operators to process user secrets. We name them Type-I and Type-II for brief, as depicted in Figure 1. Both types reuse existing database engines, inheriting (almost) all features of modern databases such as SQL execution and ACID transactions.

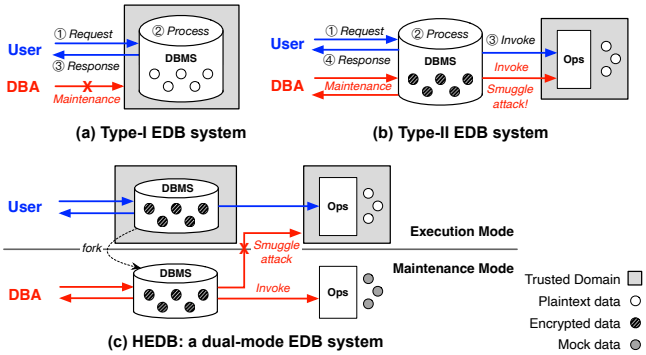


Figure 1: *Existing EDB systems can be categorized into two types: Type-I lacks maintenance and Type-II lacks interface security.* HEDB *leverages a dual-mode design to support both.*

For Type-I EDBs [17, 44, 45], a system operator or DBA can only monitor an end-to-end secure channel between an isolated database engine and a remote user. However, the conventional role of DBAs conflicts with customer privacy. Consider a maintenance task that DBAs help to diagnose a database misconfiguration bug [40]. After curious DBAs log into the database server, they can read whatever user data of interest because of their high privilege. Notably, eliminating the role of DBAs from the database engine requires non-trivial engineering efforts. Furthermore, excluding DBAs will give up the benefits of their expertise in managing, optimizing and diagnosing the outsourced databases.

Type-II EDBs [14, 15, 30, 42, 43, 48, 50] typically use database extensions to enable various primitive operators over encrypted data. Operators include arithmetics, comparisons, string searching, etc. The primary advantage of Type-II is that operators have a small trusted computing base (TCB) compared to Type-I. Furthermore, the low complexity of the operator's codebase also makes it easy to develop and simple to vet. Most importantly, Type-II surpasses Type-I in terms of

maintainability, as DBAs can connect to the database server, examine query plans, attach powerful profilers and debuggers, and collect crash dumps without concerns of data breaches, since user data is always encrypted. Type-II EDBs are therefore well adopted by cloud database vendors [14, 30, 50]. Regarding data privacy, Type-II leaks information such as ordering and frequency, which may compromise sensitive columns with the aid of sophisticated background knowledge [27–29, 32, 39].

Even worse, under existing database access control, an adversarial DBA can arbitrarily invoke Type-II EDB's operator interfaces. As Type-II exposes various operators, DBAs can exploit a sequence of carefully constructed invocations to recover the victim's sensitive data [16]. We devise an efficient and stealthy attack, named *smuggle attacks*, which applies to all basic encrypted types (i.e., numerics, time, text) and can recover 100% data items of 35,243 health records within 2 minutes with no prior knowledge. Conceptually, smuggle attacks is similar to Iago attack [21], as both abuse interfaces. But unlike Iago, defending smuggle attacks is more challenging because it does not tamper with the correctness of invocation results. Hence, we opt for a new approach to defeating smuggle attacks, while retaining Type-II's advantages of DBA maintainability.

Our proposal: HEDB[1]. HEDB is a new EDB design that can provide interface security (namely, smuggle attacks-resilient) and maintainability. HEDB's design is based on two insights: (a) without authenticated access, interface security cannot be achieved, and (b) in most cases, accessing plaintext secret data is not essential to EDB maintenance. Hence HEDB introduces two modes: *Execution Mode* where operators authenticate valid requests for user queries (defending against smuggle attacks), and *Maintenance Mode* where mock data is used during DBA maintenance (minimizing privacy leakages). In Execution Mode, HEDB adopts Type-II's design by decoupling the DBMS and operators, and protects them using two trusted domains with an authenticated channel. When switching to Maintenance Mode, HEDB forks a new DBMS instance from the protected DBMS to an unprotected domain, and feeds operators (also in the unprotected domain) with mock data. Figure 1 overviews this process.

This dual-mode EDB design is non-trivial and has several technical challenges. First, switching EDB components between modes requires execution environment reconstruction for maintenance purposes. Second, too accurate maintenance may help DBAs infer secret data easily, while simply using fake data hinders maintenance. Third, after maintenance, there should be a secure way to apply hotfixes to the protected DBMS instance, without invoking any new attack surfaces.

To overcome the above challenges, HEDB introduces several key techniques. To allow DBAs to inspect the stateful DBMS, HEDB employs DBMS-located VM fork across two hypervisors using existing hardware (i.e., ARMv8.4 S-EL2). For execution environment reconstruction, HEDB relies on record-and-replay. HEDB records the operator invocation trace in Execution Mode, and proposes *authenticated replay* to reproduce DBMS issues in Maintenance Mode. To preserve buggy control flows and protect user data privacy at the same time, HEDB proposes *anonymized replay*, which employs concolic execution to capture path constraints, translates data masking rules also into constraints, and exploits constraint solving for operator troubleshooting in Maintenance Mode. Finally, HEDB uses *maintenance templates* to securely apply hotfixes in Execution Mode. HEDB accomplishes these features with low implementation complexity (~2K lines of C and Python code). HEDB's record incurs 5.88% runtime overhead; replay supports fixing configuration bugs, reproducing functional bugs, and debugging most performance bugs. Our optimizations speed up HEDB's TPC-H execution by 2.49×, and improve HEDB's constraint solving-based log anonymization by up to two orders of magnitude.

Contributions. We highlight the following contributions:

- A study of existing EDB systems and the introduction of smuggle attacks for Type-II EDBs.
- A dual-mode EDB design, based on empirical studies of typical maintenance issues and DBA operation tasks.
- A new system called HEDB, which prevents smuggle attacks while allowing DBAs to maintain EDB with reasonable overhead.

While HEDB provides, for the first time, interface security and maintainability for existing Type-II EDB systems, it does have some limitations. HEDB's current implementation does not support non-deterministic bug reproduction (e.g., concurrent transactional writes such as in TPC-C, though TPC-C is not vulnerable to smuggle attacks). In addition, HEDB does not cover all DBA tasks (e.g., arbitrary query rewriting) and may not reproduce all bugs (when using strict masking rules). Nonetheless, HEDB fills a critical gap in encrypted databases.

2 Background and Motivation

2.1 Database as a Service (DBaaS)

In "Database as a Service" (DBaaS) [31], service providers take care of the installation, update, backup, and maintenance of databases. DBaaS provides managed databases with a transparent software stack and infrastructure. This design releases users from the duty of database administration, which is complex, time-consuming, and requires deep expertise. In DBaaS, these maintenance tasks are delegated to database administrators (DBAs). In brief, DBaaS empowers users to focus on their core business.

2.2 Encrypted Database (EDB)

Data privacy is a major concern of adopting DBaaS. Service providers might not be fully trustworthy [4]; even if they

[1] HEDB is named after He (Helium), the 2nd element, implying its two modes.

Type	EDB System	Approach	F	S	M
Type-I (TEE-based)	TrustedDB [17]	database on a secure coprocessor	●	●	○
	EnclaveDB [44]	database in Intel SGX	●	●	○
	DBStore [45]	database in ARM TrustZone	●	●	○
Type-II (Crypto-based)	CryptDB [43]	operators using crypto schemes	○	●	●
	Arx [42]	operators using crypto schemes	○	●	●
	Monomi [47]	server crypto + client computation	●	●	○
Type-II (TEE-based)	Cipherbase [15]	operators in FPGA	○	●	●
	StealthDB [48]	operators in Intel SGX	●	○	●
Type-II (TEE-based from industry)	Always Encrypted [14]	operators in Intel SGX	○	●	●
	FE-in-GaussDB [30]	operators in ARM TrustZone, Intel SGX	●	○	●
	Operon [50]	operators in Intel SGX, FPGA	●	○	●
Type-II	HEDB (this work)	dual-mode security architecture	●	●	●

Table 1: *Survey of existing EDB systems. F: Functionality; S: Security; M: Maintainability.*

are, curious staff may leak private information. For instance, Swiss bank DBAs were reported to have sold customer information [12]. This is why an encrypted database (EDB) comes into place; an EDB executes queries over fully encrypted data.

Ideally, an EDB system should provide a compatible set of traditional DBMS features (e.g., transactions, recovery) and most importantly, support all common SQL queries on the encrypted data. For example, users can perform equality checks to the highly sensitive personally identifiable information (PII) such as names and credit card numbers. As another example, users should be able to apply arithmetic operations and range predicates on the encrypted financial data (e.g., billings) and healthcare records (e.g., heart rates) to calculate the maximum expense or to compute the average heart rates.

Both academia [15, 17, 26, 42–44, 48] and industry [14, 30, 50] have shown great interest in EDB systems. We surveyed state-of-the-art EDBs as listed in Table 1, and classified them into two categories: (1) a monolithic EDB design, and (2) a plug-and-play EDB design. For simplicity, we name them Type-I and Type-II, respectively.

2.3 Type-I EDB: Putting a Database in TEE

Overview. Trusted execution environments (TEE) are a hardware-assisted approach that offers the essential abilities of secure isolation, memory encryption and remote attestation. They are widely available on commercialized processors (e.g., AMD SEV [13], Intel SGX [11] and TDX [9], ARM S-EL2 [35] and CCA [36]) or implemented using a secure co-processor or FPGA. The monolithic EDB design places an existing DBMS engine into TEE to protect user data and queries. User secrets are encrypted outside TEE and remain plaintext inside the trusted database. This design brings a large trusted computing base (TCB); an operating system or library OS must be ported into the TEE [17, 44, 45].

Workflow. A user queries the Type-I EDB as follows: ❶ The client-side user or the DB-backed application issues a SQL query to DBMS through a secure channel. ❷ DBMS parses the query, generates a plan, optimizes it and executes the plan, by reading the encrypted tables from the untrusted storage, and writing the updated tables after encryption. ❸ DBMS returns the query result through the secure channel.

Implications. In Type-I EDB systems, the data privacy and database implementations are tightly coupled, which raises several issues. First, simply putting a database into TEE does not make the database immune to rogue DBAs. For today's DBMSes, DBAs have unlimited access to users' data, including secret data in the TEE. To ensure privacy, Type-I EDBs must either modify DBMS engines or disable the role of DBAs. However, refactoring the DBMS codebase to preclude the existence of DBAs and their privileges may require significant engineering efforts. Even if a DBMS eliminates DBAs, it would give up maintainability—this DBMS loses the major benefit of DBaaS that experts (i.e., DBAs) manage, optimize, and diagnose users' outsourced databases. People might not use DBaaS in the first place.

2.4 Type-II EDB: Putting an Operator in TEE

Overview. The plug-and-play EDBs are another type of EDB. They leverage customizable extensions of modern database systems (e.g., PostgreSQL, MySQL) to encrypt data on-the-fly. The extension is written as a database plugin (normally in the form of a user-defined function or UDF). To implement Type-II EDBs, developers typically create and register new data types—encrypted data types—into the database. When the database execution engine processes encrypted data types, it invokes the UDF-based operators that are responsible for handling encrypted data operations.

There are two ways to implement Type-II EDBs. For one, developers implement different cryptographic schemes in operators to compute directly on the encrypted data [42, 43, 47]. We call them crypto-based Type-II EDBs. For the other, developers implement operators in TEEs. We call these TEE-based Type-II EDBs [14, 15, 30, 48, 50]. TEE-based EDBs rely on hardware modules to provide integrity and confidentiality, and data are decrypted only when they are within TEEs. Crypto-based Type-II EDBs fall short in functionalities (e.g., floating-point arithmetics and text concatenation); they must either rely on a trusted proxy [42, 43] or move the unsupported computation to the client [47]. In this paper, we focus on the TEE-based Type-II EDBs that have full-SQL supports and are preferred in production [14, 30, 50].

Workflow. A user queries the Type-II EDB as follows: ❶ The client-side user or the DB-backed application sends a SQL query whose sensitive constants are encrypted. ❷ DBMS parses the query, and reads the encrypted data from storage. The DBMS engine generates, optimizes, and executes an execution plan. Upon each computation of the encrypted data type, DBMS prepares a tuple, ⟨`ciphertext1, ciphertext2, ...`⟩, and feeds it to the operator. ❸ The operator receives the tuple, decrypts the ciphertexts, performs the operation, encrypts the result (except when returning plaintext boolean values, e.g., comparisons), sends the result to DBMS, and waits for the next invocation. ❹ The DBMS

engine finishes the entire query execution by returning the (encrypted) result to the client.

Advantages. In comparison with Type-I EDBs, the Type-II EDBs have the following advantages.

- *Small TCB*: Compared with putting a full-fledged DBMS in TEE, Type-II EDBs run only operators in TEE which is a tiny fraction of the entire DBMS.
- *Development friendly*: The Type-II EDBs leverage DBMS extension systems and require no modifications to DBMS engines. The low complexity of operators also makes upgrades simple and easy.
- *Maintenance friendly*: The DBMS engine does not touch plaintext data, so it is accessible to DBAs. DBAs can perform maintenance operations such as examining query plans for performance, collecting crash core dumps for troubleshooting, or even attaching a debugger to inspect the execution of a SQL query.

These advantages make Type-II EDB preferable to cloud vendors such as Azure [14], Huawei [30] and Alibaba [50].

Implications. Compared to Type-I, Type-II EDBs however expose a larger attack surface. First, unlike Type-I, Type-II does not protect the integrity of query execution as it relies on an unprotected DBMS engine. Second, the data-level computation allows an honest-but-curious DBA to learn the data volume, distribution, frequency, ordering, and correlations between columns. With prior knowledge, an adversary may be able to infer secret data [27–29, 32, 39]. Finally, if a malicious DBA can issue arbitrary operator invocations, they can conduct a full database breach. We call it *smuggle attack*.

2.5 Smuggle Attack

This section describes how a DBA can mount a smuggle attack to recover encrypted data types and real-world datasets. We emphasize that the smuggle attack requires no background knowledge, and its recovery is deterministic.

Attack overview. We use a minimal working example that recovers encrypted integers in a Type-II EDB.

- *Constructing basic ciphers*: By division ($\div$), a DBA can obtain the ciphertext of '1' (dividing a number by itself). With the basic ciphers of '1', in principle, the DBA can construct all encrypted integers by iteratively asking operators to add ($+$) the cipher '1' to a counter.
- *Recovering user secrets*: With the equality operator ($=$), the DBA can recover the victim's encrypted values by observing the plaintext boolean values by comparing them with known ciphertexts. To recover an encrypted integer x, the DBA can use a binary search to compare x with some candidate known ciphertexts (using $<$, $>$, and $=$).

Other encrypted types (e.g., decimal, text, and time) can also be attacked (see § A.1). Extending their data domain to a larger range (e.g., 64-bit) does not prevent the attack because binary search is efficient to search on even a 64-bit range.

System	Example	API numbers	Interface attack
Kernel	Linux	200+ POSIX APIs	Iago attack
DBMS	PostgreSQL	79 operator APIs	Smuggle attack

Table 2: *The analogy between Iago [21] and smuggle attacks.*

Removing operators used by smuggle attacks will disable OLAP workloads because these workloads (e.g., TPC-H) require all the mentioned operations (e.g., $\div$, $+$, $>$, $=$).

Attacking real-world datasets. We illustrate smuggle attacks against a real-world dataset, SPARCS[2], with 2.54 million records [8]. We use an open-source Type-II EDB, StealthDB [48] (commit `1ca645a`), which exposes operators such as arithmetics, comparisons, mathematics, aggregations ($+$, $-$, $*$, $/$, $\%$, $<$, $=$, `power()`, `MAX`, `AVG`, `SUM`). Only comparison operators return boolean values in plaintext; others return the computation results in ciphertext.

We first select 6 columns of SPARCS patients' sensitive information from 239 hospitals in 9 areas, and protect these columns using StealthDB with the AES-128-GCM encryption. We then log into StealthDB using a DBA account and can call operators with crafted parameters, but cannot see the internals of operators (i.e., cannot see decrypted user data). Lastly, we issue binary-search SQL queries to conduct smuggle attacks; these queries do not return to users nor impact their queries' results. In the end, smuggle attacks recover 100% ciphertexts in 92 seconds without any prior knowledge.

Defending smuggle attacks is challenging. We argue that smuggle attacks are hard to defend by today's EDB designs. This is because smuggle attacks are an interface attack that targets the exposed operator interfaces, rather than any particular implementations. We have seen interface attacks before, for example, Iago attack [21] that targets OS interfaces (i.e., system calls). We summarize the two attacks in Table 2.

In fact, defending smuggle attacks is even harder than preventing Iago attack because Iago attack can be identified by checking if a syscall follows its specification. For example, the return value of `sbrk()` must not fall into any range of the allocated memory areas; otherwise, there is a data corruption (and this is likely an Iago attack). Unlike Iago attack that is conducted by few syscalls (usually just one syscall), smuggle attacks require a series of invocations. Neither operators nor users can resist smuggle attacks because (1) operators within TEEs cannot distinguish user's invocations from others (e.g., malicious DBAs); (2) invocations issued by smuggle attacks do not alter the correctness of user queries, and hence users cannot realize that a smuggle attacks is happening.

Attack summary. The core principle behind smuggle attacks is not new, as it has been established that any column that enables both computation and comparison operations could be vulnerable (as noted on page 6, "Write query ex-

[2]The dataset we use does not contain protected health information (PHI) under Health Insurance Portability and Accountability Act (HIPAA); all data elements considered individually identifiable have been redacted.

ecution" in [43]). However, we are the first to successfully apply this principle to a real-world EDB system. Prior EDB attacks have identified several types of leakage attacks, such as Count Attack [19], Non-Crossing Attack [29], Access Pattern Attack [32], and Frequency Analysis [39], which are also applicable to both Type-I and Type-II EDBs. What sets smuggle attacks apart is that it requires zero prior knowledge, making it even more potent than previous leakage attacks. Additionally, smuggle attacks are not exclusive to DBAs, as anyone who can access operator interfaces can carry out smuggle attacks. For instance, an attacker who knows a victim's password but not the encryption key could not decrypt the victim's data, but they could bypass access control with the password and then use smuggle attacks to breach the data.

We have studied Type-I and Type-II EDB system designs, where there is tension between (a) database maintenance and (b) interface security. In short, Type-I is immune to (b) but lacks (a), while Type-II provides (a) but suffers from (b). However, both (a) and (b) are essential for EDBs; we need both. This motivates our system, HEDB.

3 HEDB Design

We first introduce our design goals and present a new EDB architecture that HEDB uses. We then describe our threat model and how HEDB works.

Design Goals. HEDB has three goals.

- *G1: smuggle attack resilience.* HEDB must protect user's sensitive data from smuggle attacks (§ 2.5) which Type-II EDBs [14, 30, 50] suffer from.
- *G2: database maintainability.* A DBA should be able to configure, manage, diagnose, and troubleshoot the HEDB as a traditional DBMS.
- *G3: backward compatibility.* HEDB aims to be compatible with the existing database ecosystems. We do not expect to reimplement HEDB in new frameworks (e.g., verifiable computation [51] or secure multi-party computation [41]) which invalidates existing DBMS tools.

HEDB architecture. HEDB uses a new three-zone architecture (depicted in Figure 2) because we observe that different roles—DBAs, DBaaS providers, and database users—have different duties and requirements. (1) DBAs are responsible for managing resources and performing maintenance tasks, and they want to do these jobs in a low-drama way. (2) DBaaS providers are supposed to deploy DBMS as services, and they want their services running correctly. (3) Users are the data owners whose secret data is stored in the database. They want the database to be easy to use (e.g., maintained by DBAs), and meanwhile, users need their data stored securely (i.e., having data integrity and confidentiality).

These observations inspire HEDB's architecture: unlike prior EDBs [17, 44], HEDB ***decouples integrity from privacy***. In particular, HEDB's architecture detangles DBAs' maintenance jobs from users' data confidentiality requirements by using three zones: *integrity zone*, *privacy zone*, and *management zone*. The integrity zone provides execution integrity but not confidentiality; it runs the DBMS engine. The privacy zone guarantees data confidentiality; it runs operators and is the only place containing users' plaintext data. The management zone allows DBAs to troubleshoot both DBMS engine and operators. This design brings the opportunity to serve both interface security (*G1*) and maintainability (*G2*).

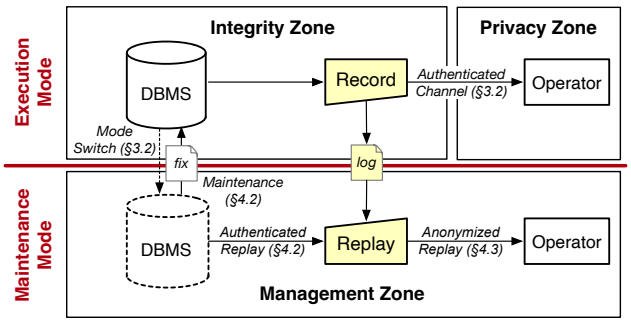


Figure 2: HEDB*'s high-level architecture.*

Threat model and security guarantees. HEDB assumes TEE hardware works as expected; that is, hardware isolation and security guarantees are reliable and trustworthy. Further, HEDB assumes remote attestation for authentication. HEDB uses remote attestation to confirm the integrity of the EDB executables. We also assume that database users and DBaaS providers agree on the EDB code and configurations.

In HEDB's threat model, DBaaS providers are not trusted, as they could access server-side states over the network, on disk, or in memory that are not protected by TEE. They may also tamper with the database logs and data, and drop network connections to the EDB systems. These attacks can be detected by HEDB. Likewise, cloud administrators (who manage the cloud's physical resources) and database administrators (DBAs) are not trusted either and can behave arbitrarily, including conducting smuggle attacks. Conversely, HEDB assumes that users will not intentionally attack themselves or leak their own data. However, co-tenant users may pose potential threats and they can be blocked using the DB's access control. Finally, the developers of HEDB are trusted, but the source code must be verified. Thanks to the small pieces of code in Type-II operators, HEDB is made easy to verify.

As security guarantees, HEDB ensures no plaintext data outside TEEs, the same as Type-I and Type-II EDBs. In addition, HEDB is smuggle attacks resilient. In terms of metadata privacy (e.g., frequency, ordering), HEDB provides the same security guarantees as Type-II EDBs. Both HEDB and Type-II EDBs may leak metadata [27]. Nonetheless, this is a fundamental trade-off between functionality and privacy because revealing these metadata is sometimes necessary for core database functionalities, for example, database indexing. Production systems have made the trade-off by choosing Type-II EDBs as the de facto method [14, 30, 50]. Finally, similar to

both Type-I and Type-II EDBs, HEDB does not prevent exploitations of DBMS bugs or vulnerabilities (e.g., code-reuse attacks), which is an orthogonal line of security problems.

3.1 HEDB Workflow

HEDB provides two modes: *Execution Mode* and *Maintenance Mode*. Execution Mode is when HEDB normally runs. HEDB serves user queries by running the DBMS engine in the integrity zone and executing operators in the privacy zone. When performing maintenance, DBAs switch HEDB to Maintenance Mode, in which operators stop responding to new requests and DBMS is forked to the management zone. Management zone enables DBAs to inspect the internal states of the DBMS engine. After locating issues and suggesting solutions, DBAs switch HEDB back to Execution Mode and resume the service.

Normal execution. To launch a HEDB instance, a hypervisor starts a virtual machine (VM) in the integrity zone, and runs a DBMS instance in the VM. The hypervisor calculates the digest of the VM and ensures its integrity. Meanwhile, the hypervisor initializes operators that run in the privacy zone.

After HEDB's initialization, a user can remotely attest both HEDB's VMs (containing DBMS and operators). Then, the user establishes a secure channel with the DBMS instance and starts sending queries. Note that the query constants are encrypted. For example, in a query `SELECT ... WHERE year < 2022`, the number 2022 will be encrypted. This is a must because the integrity zone does not provide confidentiality.

In Execution Mode, DBAs are isolated from the HEDB. DBAs cannot log into the database VM or access operator interfaces hence cannot start attacks. HEDB achieves this by disabling the logins for VM superusers and DBA accounts when booting. Users can verify this by checking the booting script and attesting that the script is the one that runs.

To monitor resources while HEDB is running in Execution Mode, VM resources can be externally monitored by the cloud hypervisor, and DBMS resources can be queried using statistics SQLs via a normal user (i.e., non-DBA) account.

Database maintenance. When users encounter problems, they seek DBAs for assistance. DBAs can request HEDB to switch to Maintenance Mode. HEDB does this by forking the current DBMS engine and dumps two logs, *authenticated log* (§4.2) and *anonymized log* (§4.3). In Maintenance Mode, HEDB uses record-and-replay [24] to help DBAs run user queries. The record-and-replay enables DBAs to profile, diagnose, and troubleshoot EDB in the management zone. We elaborate on how HEDB supports maintainability in section 4.

After troubleshooting, DBAs submit a fix and request HEDB to switch back to Execution Mode. During switching, HEDB in the integrity zone examines the fix (§4.2). HEDB will reject if the fix does not pass the check or DBAs tamper with the code or the (encrypted) data of the database.

Mapping HEDB architecture to real hardware. HEDB makes some security assumptions about the hardware. For example, HEDB requires the privacy zone to provide either memory encryption or dedicated on-chip memory. In fact, HEDB's architecture can be achieved by using today's hardware. The current HEDB prototype relies on commercial-off-the-shelf ARMv8.4 S-EL2 using a Normal World VM as the management zone, a Secure World VM as the integrity zone, and another Secure World VM with on-chip memory as the privacy zone. Both management zone and integrity zone support virtual machines (VMs) atop hypervisors [35]. It can be further extended to the next-generation confidential computing platforms such as Intel TDX [9] (using a Normal VM as management zone, a TD VM as integrity zone, and an SGX enclave as privacy zone) and ARMv9 CCA [36] (using a Normal VM as management zone, a TrustZone VM as integrity zone, and a Realm VM as privacy zone). While HEDB is designed for virtualized environments, its solution does not intrinsically rely on the VM. For bare-metal systems, self-migration [34] can be used as an alternative.

3.2 Defending Smuggle Attack

Existing commercialized Type-II EDB products [14, 30] defend smuggle attacks by sacrificing functionalities. For example, Azure AEv2 [14] does not provide arithmetic operations, and Huawei FE-in-GaussDB Production [30] does not provide comparison operations. Neither of them can support analytical queries such as TPC-H. Alibaba Operon [50] is the first system that supports full-SQL operations but restricts the callee by specifying which operators can be invoked. Nonetheless, when users need to execute TPC-H, Operon then fails to stop smuggle attacks because TPC-H contains both arithmetic and comparison operators, and attackers can use them too. Instead, HEDB chooses to restrict the caller by authenticating the invoker (described below); HEDB prevents DBAs from invoking any operators. Such a design enables diverse operators without any concerns regarding interface attacks.

Defending smuggle attacks by mode switch. HEDB prevents smuggle attacks by switching the DBMS engine from Execution Mode to Maintenance Mode; a DBA cannot access the DBMS engine in Execution Mode and cannot invoke the operators in Maintenance Mode. Regarding mode switch, HEDB chooses to fork VMs rather than processes, and furthermore, many DBMS engines use multiple processes. Forking a group of processes requires forking their OS kernel states in addition to careful synchronization (to avoid deadlocks). Besides, trouble may arise from the kernel, such as insufficient buffer cache and limited process number (see Table 4). As a result, we choose to fork the DBMS-located VMs instead of the DB processes, since both management zone and integrity zone support hardware virtualization. Our design choice is simple and practical, and meets our goals (*G1, G2, and G3*).

Defending confused deputy by authenticated channel. In modern databases, a database user can use the SQL command

Intention	Operation
monitor waiting sessions	rank running sessions from `pg_stat_activity`
monitor waiting threads	rank running threads from `pg_thread_wait_status`
monitor database locks	analyze lock situations from `pg_locks`
identify slow queries	analyze SQL statements from `pg_stat_statements`
explain database plan	issue `EXPLAIN [SQL statement]`
collect database statistics	issue `ANALYZE [table]`
Example-1: query waiting events of the current running sessions	`SELECT wait_event, wait_event_type, Count(*) FROM pg_stat_activity GROUP BY wait_event, wait_event_type ORDER BY Count(*) DESC;`
Example-2: query transactions that start longer than a specified duration (100s)	`SELECT Count(1) FROM pg_stat_activity WHERE pid != pg_backend_pid() AND (Now() - xact_start > interval '100s');`

Table 3: *The intentions and the corresponding DBAs' operations for* ***Step-1 inspections*** *(PostgreSQL-based EDB). The observed phenomena and subsequent actions are listed in Table 4.*

"`SET ROLE`" to change the user ID of the current session. DBAs can thus switch to any user to launch the smuggle attacks. In HEDB, we adopt a client-side authentication technique. The client must hold a master key, and the operators in the privacy zone can remotely attest to the client using standard signature verification. Because DBAs do not have the user credential, an operator rejects requests from the DBAs, even when the session has the user ID. Our survey shows that existing commercialized EDBs [14, 30, 50] all support client-side encryption where the client holds a master key.

4 Supporting Maintainability

HEDB is designed to support database maintainability. For HEDB (or any Type-II EDB), there are two major pieces that require maintenance and troubleshooting: the DBMS engine and operators. By studying DBA daily tasks (§4.1), we observe that DBAs operate differently on the two parts and expect different tools and functionalities. HEDB supports DBMS engine maintenance through authenticated replay (§4.2) and operator troubleshooting through anonymized replay (§4.3).

4.1 Understanding DBA tasks

To understand database maintenance, we conduct an empirical study of existing DBA guidance from Microsoft SQL Server, MySQL, PostgreSQL, and several cloud databases, including Amazon Aurora [1], Google Cloud SQL [7], Azure SQL [14], Huawei GaussDB [30], and Alibaba Operon [50].

We find that the workflow of DBA administrative tasks typically contains two steps. In **Step ❶**, DBAs inspect the states of DBMS engine and OS to identify the issue and locating the root cause (see Table 3). During inspection, DBAs may need to install and use profiling tools or issue proper SQLs to query various database metadata tables (e.g., index, locks, activity), for example, examining transactions that last longer than a desired duration, say 100 seconds. In **Step ❷**, DBA takes actions to fix the issue (see Table 4). These actions mainly involve updating the configuration parameters of the DBMS engine or the underlying OS kernel, kill the deadlocked database processes, or reclaim database storage.

Observations to support maintainability. We have two observations from the above two-step maintenance process. The first observation is that inspections (**Step ❶**) can be arbitrary and complex, while the action-taking (**Step ❷**) is rather regular and structured. We therefore allow DBAs to conduct any necessary inspections on the forked DBMS engine in Maintenance Mode (these inspections do make temporary changes but can be discarded), and provide a *maintenance template* which translates maintenance actions into a finite whitelist of tasks in Execution Mode. Second, we observe that for operators, DBAs need to reproduce the control flow in order to trigger bugs, but do not necessarily need the original inputs (i.e., user secrets). Hence, HEDB provides a set of fake inputs that preserve operators' control flows.

4.2 DBMS Maintenance by Authenticated Replay

In this section, we introduce how HEDB supports DBAs to maintain the DBMS engine. We describe operator troubleshooting in the next section (§4.3).

Overview. When meeting problems, users contact DBAs for help. DBAs will request HEDB for a mode switch from Execution Mode to Maintenance Mode. In Maintenance Mode, DBAs fork the VM without worrying about accidentally damaging the VM snapshot. In the cloned VM, DBAs have the root privilege and can re-execute the problematic user requests by authenticated replay (described in detail below). During troubleshooting, DBAs can use arbitrary tools (e.g., profilers and debuggers). There are no privacy leaks during troubleshooting because user data is encrypted in the VM.

After the root cause is identified, HEDB provides a *maintenance template* where DBAs can write the actions to be applied. Then HEDB is switched to the Execution Mode. The integrity-zone hypervisor triggers a shim module in the VM. This shim first performs sanity checks over the submitted fix, ensuring all parameters in the template are valid, and ultimately takes the maintenance actions on the DBA's behalf.

Authenticated replay for Step ❶. To provide DBA maintenance without letting DBAs access operator interfaces, HEDB records the operations' inputs and outputs in ciphertexts during Execution Mode, and replays them to mock operator executions in Maintenance Mode. On the one hand, authenticated replay rejects any new operator invocations with unseen parameters, stopping smuggle attacks. On the other hand, authenticated replay ensures that the database follows the same control flow and data flow as in the history of Execution Mode. Using authenticated replay, various DBMS bugs (e.g., configuration and functional bugs) can be fully reproduced with the replay log. The log also embeds a timestamp of each operator invocation, and HEDB provides delay simulation to help debug performance bugs. For example, a DBA can re-execute the user queries after updating a configuration parameter and check if this update does improve the query performance.

Phenomenon	Action	Maintenance Template	Sanity Checks
OS			
directory permission denied	change directory permission	`chmod 750 [dir]`	[dir] must be under "/usr/local/pgsql/data"
coredump makes no space left	remove coredump file	`rm /var/crash/*.core`	NONE
slow buffer cache	enable huge pages for shared_buffers	`hugepage [on|off] [num]`	[num] is between 64 and 65536
Connectivity			
DB connection failure	restart DB engine	`systemctl restart postgresql`	NONE
too small MTU	reconfigure network card's MTU	`ifconfig [eth] mtu [num]`	[eth] exists and [num] is between 64 and 8192
"sorry, too many clients already"	enlarge the process number	`ulimit -u [value]`	[value] is between 1 and 8000
Database			
"No space left on device"	vacuum the database	`VACUUM FULL;`	NONE
index contains corrupted page	rebuild the index	`REINDEX TABLE [table];`	[table] must be an existing table
too large log files	remove unused log files	`SELECT pg_rotate_logfile();`	NONE
a query is hung or blocked	cancel the hung query	`SELECT pg_cancel_backend([pid]);`	[pid] must be an active database process
low throughput	adjust I/O load of background writer processes	`max_io_capacity` = [num]	[num] is between 0 and 100000
lock wait timeout	enlarge timeout values	`max_query_retry_times` = [num]	[num] is between 0 and 3600
insufficient buffer	update DB buffer configuration parameters	`shared_buffers` = [num]	[num] is between 64 and 2048

Table 4: *The typical phenomena and DBAs' common* ***Step-2 actions****. The operations used to observe these phenomena are listed in Table 3.*

Maintenance templates for Step ❷. HEDB translates common actions into templates. For example, to adjust the transaction timeout, a template is "`max_query_retry_times = [num]`", where the "`[num]`" is a parameter that DBAs fill in and is restricted to a reasonable range (between 0 and 3600 seconds). We summarize common DBA actions and the corresponding templates in Table 4.

Maintenance templates offer a quick path to implement DBA hotfixes. Our lessons with template-based maintenance show that it covers common DBA actions used in practice, such as updating the configuration parameters, fine-tuning slow queries, and canceling lengthy transactions. For actions that require modifying the database code, such as patching functional bugs or adding new query-rewrite rules to the DBMS engine for better performance, HEDB requires auditing the patch before updating.

4.3 Operator Troubleshooting by Anonymized Replay

As operators are highly extensible and designed to support various operations, bugs are inevitable. Unlike the DBMS engine that only handles ciphertexts, operators work in the privacy zone that contains user secrets in plaintext. Debugging operators requires avoiding or minimizing data leakage.

Overview. The core idea is to construct "control-flow equivalent" inputs using a concolic executor and a constraint solver. This process generates multiple sets of inputs, causing the operator to exercise the same path as the buggy inputs. HEDB selects a new set of inputs from candidates, replaces the encrypted values of the authenticated log (called *anonymized log*), and replays the log to reproduce the operator's bugs.

While [20, 22, 49] have also used similar techniques for diagnosis under privacy regulation, they suffer from issues related to path explosion or environment modeling. HEDB enhances these techniques, improving efficiency and privacy.

Efficient constraint collection via simplified operators. HEDB overcomes the efficiency challenges in three ways. First, operators are userspace programs with rare system calls (mostly memory allocation) and no privileged instructions, hence eliminating the need for modeling OS kernel environments. Second, operators are designed to be stateless, which is common in Type-II EDBs [14, 30, 50]. This means that an operator's path conditions rely solely on its inputs, resulting in significantly fewer possibilities. Third, when operators become complex, scalability issues with concolic executors might limit their practicality. HEDB requires developers to decompose complex operators into micro-operators, each of which should undergo the concolic executor within a reasonable short amount of time.

Privacy-preserving log generation via data masking. To hide user data, HEDB requires a large number of distinct candidates sharing the same control flow as the original user data, which is computationally expensive. For example, generating 1 million candidates for a single control flow takes 24 minutes using a state-of-the-art constraint solver, Z3 [23]. While increasing the number of candidates enhances privacy, determining the optimal number of candidates is non-trivial. To address this, HEDB leverages rules from modern data masking engines [2, 5]. Common rules include scrambling (`1234` → `XXX4`), substitution (`Boston` → `USA`), variance (`0.07` → `1.0`), etc. Currently, HEDB employs simple rules translated into constraints understood by Z3.

By feeding both path constraints and masking constraints into the constraint solver, HEDB can generate new inputs that *not only reproduce the bug but also protect user privacy*. This generation only needs to occur once per control flow, and the results can be reused for later debugging.

The input parameters of operators comprise two types: user data (e.g., from columns) and metadata (e.g., size). Following the security model of previous work [42, 48], HEDB masks only user data, while metadata can be hidden using padding. Users can customize the data masking rules for different columns based on their knowledge of the data semantics.

Example. Here is a demonstrative example of how HEDB generates the anonymized log. As illustrated in Figure 3, each entry in the authenticated log is iteratively translated into an anonymized counterpart. First, a line of log entries

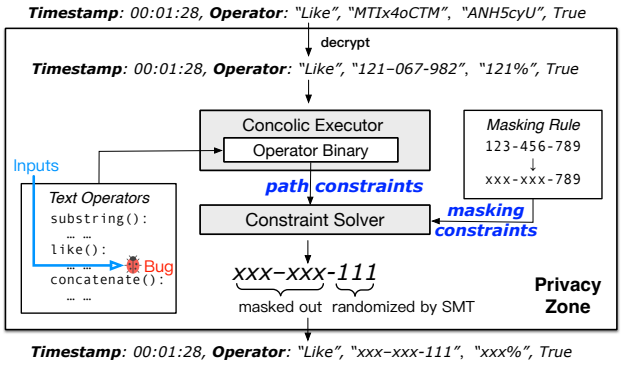


Figure 3: *How* HEDB *translates an authenticated log entry into an anonymized log entry for replay-based operator troubleshooting.*

is decrypted in the privacy zone, and fed into a concolic executor that captures the path constraints that lead to the "`like`" operator using buggy inputs. Then, an constraint solver utilizes these constraints, along with those derived from a rule that scrambles the first 6 digits of a phone number, to generate a new set of fake inputs, namely, "`XXX-XXX-111`".

Design rationale. In principle, HEDB's troubleshooting is not limited to stateless operators. However, supporting stateful operators requires overcoming additional challenges. First, crash consistency is a critical issue for Type-II plus stateful operators, because failures can cause inconsistencies between the states of the DBMS and the operators, and is inherent to all Type-II systems regardless of HEDB's design. Second, consider HEDB with stateful operators; concolic execution might experience state explosion, whereas record-and-replay will need to log every state change, resulting in performance degradation. Finally, applying HEDB's data masking rules to operator states may raise security concerns, since these states are typically less structured and could potentially reveal information. Another question to ask is whether the proposed approaches could be applied to provide maintainability to Type-I EDBs. This is an open question, as it presents significant obstacles, such as (i) the challenge of anonymizing the DBMS's intricate internal states, and (ii) the potential inability to scale over extensive execution paths, given the current concolic executors and constraint solvers.

5 Implementation

5.1 Implementation Complexity

DBMS and operators. Similar to prior Type-II EDBs [14, 30, 48, 50], we implemented an ARM version of UDF-based operators using ~4K lines of C for PostgreSQL v13.8. Our UDFs define 4 encrypted data types and 79 operators. To protect the DBMS engine in an integrity zone, we run it in a secure VM on top of a thin ARMv8.4 S-EL2 hypervisor—S-visor [35]. We further protect HEDB's operators in another secure VM with on-chip memory. We also extended S-visor to allocate a dedicated shared memory between the DBMS-VM and operator-VM, accomplishing *authenticated channel*.

Mode switch. We extended S-visor and KVM using 91 lines of C and 24 lines of ARM Assembly to implement HEDB's mode switch, by means of VM migration between TrustZone and Normal World. Specifically, HEDB configures the TZASC control registers to specify whether a VM belongs to TrustZone or Normal World, providing fast VM migration (~68K cycles) without incurring VM memory copying.

We follow the design principle of S-visor which delegates most functionality to N-visor (i.e., QEMU/KVM), while S-visor focuses on simple tasks such as saving and restoring VM contexts and carrying out necessary security checks. Instead of implementing `fork` in S-EL2, we leverage a "switch-and-fork" approach that reuses QEMU's mature features such as VM snapshotting. Specifically, a mode switch is triggered when DBMS is in Execution Mode, and N-visor signals S-visor to mark all VM memory as non-secure by updating TZASC. Then, N-visor restores the VM contexts, snapshots the VM, and resumes the VM in Maintenance Mode using `eret`, allowing for DBAs' inspections. We also extended S-visor to perform VM runtime attestation using SHA-256.

Record-and-replay. The record-and-replay is implemented using ~1.8K lines of C and Python. The authenticated logs reserve all computation results such as arithmetic operations to avoid the problem of random encryption (i.e., AES-GCM with nonce). As HEDB does not modify the DBMS engine, it cannot enforce execution determinism such as transaction ordering. Consequently, HEDB's authenticated replay does not support concurrent transactional writes (e.g., TPC-C).

For anonymized replay, we use KLEE [18] to collect path constraints of operators, and manually write data masking constraints in Python based on four masking rules. Currently, HEDB's anonymized replay does not support floating-point numbers, a limitation of the official KLEE, which can be mitigated via a variant version, KLEE-Float [37]. Z3 [23] is used as the constraint solver. To remove KLEE and Z3 from the online TCB, we run them on a stand-alone server with privacy zone support.

5.2 Optimization

Optimizing authenticated replay. The log size can become large due to substantial operations within a single query. We thus compress these logs with `gzip`. To ensure optimal spatio-temporal efficiency, we divide log entries into groups, and pipeline the log replaying on the current group and the log decompression in the next group during authenticated replay.

Optimizing anonymized replay. We adopt four optimization strategies. First, we modify KLEE by adding `fork()` to reuse its states, resulting in a warm start for KLEE processes instead of creation upon every operator invocation. Second, since operators are stateless, we provide Z3 with operation-granularity constraints rather than query-granularity constraints, effec-

tively reducing Z3's exploration costs. Third, we employ a cache to reuse Z3's generation efforts for the same constraints. Last, we exploit precomputation to detach the entire log generation from interactive troubleshooting, e.g., using `gdb`.

We explain HEDB's query execution optimizations in § A.2.

6 Evaluation

We evaluated HEDB to answer three major questions:

- What DBA tasks does HEDB support? (§6.1)
- Can HEDB protect itself from attacks? (§6.2)
- How much overhead does HEDB incur? (§6.3)

Experimental Setup. We use two evaluation platforms:

- **ARM Fixed Virtual Platform (FVP).** FVP is a cycle-accurate full-system ARM simulator used for functional correctness evaluation. We validate the design of HEDB, particularly, the correctness of mode switch on FVP.
- **ARM Kunpeng-920 Platform.** The platform is a 96-core ARMv8.2 CPU (2.86 GHz) server with virtualization host extension (VHE) support. Like prior work [35], we add the worst-case latency (8K cycles measured on FVP) to KVM upon each VM exit and each hypercall to simulate the overhead caused by S-EL2.

Testbed. The experiments are conducted using 2 KVM-enabled QEMU virtual machines running Linux 5.4.0. The integrity-zone VM runs PostgreSQL v13.8 with 32-core vCPU and 32GB memory, whereas the privacy-zone VM runs the operators with 8-core vCPU and 8GB memory. The host machine provides a 96-core ARMv8.2 CPU (2.86 GHz), 256GB memory and 512GB SSD running Ubuntu 20.04 LTS.

Workload. We focus on online analytical processing (OLAP) workloads because OLAP involves more types of operators that can lead to smuggle attacks. Previous Type-II EDB systems [14, 30, 48, 50] are unable to support OLAP securely. Due to ethical issues, we were unable to obtain real-world traces for our evaluation. Nevertheless, based on our observations, TPC-H is representative enough for realistic financial workloads. We set the TPC-H scale factor to 1 and encrypt all data types (i.e., numeric, date and text) in the schemas. We report the median query runtime in 10 runs.

6.1 Functionality Evaluation

Our study was conducted in partnership with Alibaba Cloud, a top three cloud company providing global database services in dozens of countries with more than 80 zones, all hosted on virtual machines. We worked closely with a team of over 50 DBAs who had 3 to 10 years of experience in areas including database development, database operations, and maintenance management. Their feedback confirmed our observations, insights, and taxonomy of DBA maintenance tasks.

The DBA tasks were summarized based on an analysis of 28,000 tickets collected between May 2022 and October 2022, each ticket representing a real DB issue assigned by users. This analysis provides an empirical understanding of the common daily issues faced by DBAs. We categorized these tasks into control-plane (i.e., managing DB instances) and data-plane (i.e., managing data in DB instances).

Maintenance Taxonomy	HEDB	Approach
Control-plane Management		
start, stop, backup, replica	✓	maintenance mode
configure access control policy	✓	maintenance mode
resolve failed high-availability	✓	maintenance mode
migration, switchover	✓	fast mode switch
update, upgrade	✓	explicit auditing
Data-plane Troubleshooting		
healthcheck DBMS status	✓	maintenance mode
explain plans	✓	maintenance mode
cancel hung queries	✓	maintenance template
Data-plane Tuning		
update configuration	✓	maintenance template
reindex encrypted columns	✓	maintenance template
rewrite user queries	★	authenticated replay
Data-plane Bug Reporting		
core dump DBMS crash	✓	maintenance mode
reproduce DBMS bugs	✓	authenticated replay
reproduce operator bugs	✓	anonymized replay

Table 5: *How DBAs maintain* HEDB. ★ *denotes that only rewritten queries that do not generate new operations can be executed.*

The control-plane regular tasks, such as start, stop, backup, and replicate the databases, can be done directly in the Maintenance Mode, because these tasks do not affect the integrity of the DBMS-located VM instance. In particular, HEDB provides a fast mode switch for switchover upon failures. Other control-plane diagnosis tasks, such as resolving service unavailability caused by misconfigured access control policies, or failed high-availability routines, can also be performed in Maintenance Mode. By design, HEDB supports all control-plane maintenance tasks. One exception is that DBAs may update or upgrade the EDB, which requires an explicit audit[3].

For data-plane maintenance, we categorize three classes:

- *Troubleshooting*: these tasks mainly locate the sources of service disruption, for example, by performing status checks, identifying misconfigurations, or explaining slow queries. DBAs can perform them in Maintenance Mode.
- *Tuning*: To resolve these identified problems, DBAs need to perform further tasks to tune the database, e.g., by updating configurations, canceling hung queries, rebuilding indexes or rewriting queries as a more involved procedure. Using authenticated replay, HEDB can support most of the tuning tasks if no extra operations are needed.

[3] A possible audit workflow is as follows: once HEDB users agree to update the EDB, the DBaaS provider releases the patch. Next, users or trusted third parties review the patch, and agree on the binary after a deterministic compilation. Finally, the patched EDB is launched and attested via TEE.

- *Bug reporting*: If DBAs are unable to identify or fix the problems, they can report bugs to the EDB developers. HEDB lets developers obtain the DBMS coredump, and offers replay to reproduce DBMS's and operator's bugs.

We systematically summarize DBA tasks as shown in Table 5. Next, we highlight some common use cases in detail.

6.1.1 Case Studies of DBMS Maintenance

Fixing configuration bugs. Modern commodity DBMS engines consist of various parameters that result in significantly large configuration spaces. In this case study, a DB-backed application developer reports a performance issue to a DBA seeking assistance. The DBA then switches the database from Execution Mode to Maintenance Mode and conducts intensive checks on the forked database instance. Eventually, an insufficient buffer is identified, and the DBA submits an action specifying "`shared_buffers` = 512MB". After switching back to Execution Mode, the buffer size is validated and updated from 128 MB to 512 MB. As a result, the query throughput is improved by 1.3×.

Rebuilding user indexes. When user indexes are unexpectedly corrupted or bloated, DBAs should rebuild them. In the first case, DBAs wish to reconstruct the index after vacuuming obsolete or duplicated records to reduce space consumption. In Maintenance Mode, HEDB leverages the ordering information from record-and-replay logs to assist DBAs in rebuilding the index successfully. In another case, DBAs have changed a storage parameter (e.g., `fillfactor`) for an index and want to ensure that the configuration update has taken full effect. To this end, DBAs use the maintenance template not only to alter the storage parameter but also to rebuild the indexes.

Cancelling hung queries. When EDB users experience hung queries and are unable to cancel them, they also seek help from DBAs. There are several reasons why queries may hang, all of which can be diagnosed and remedied using HEDB. First, if there are too many concurrent connections that exceed the capacity of the database service, DBAs can utilize HEDB's template to adjust the configuration parameter (e.g. `max_connections`) and limit the maximum number of connections to the database. Second, if lock contention or deadlocks exist in the database, DBAs can use an OS command through the template to send a signal to kill the process, or update the configuration parameter (e.g. `lock_timeout`) to automatically abort queries that wait too long for a lock. In the last scenario, if the database is in a recovery state, users must wait until the process is complete. However, DBAs can use a template to update the configuration parameter (e.g. `idle_in_transaction_session_timeout`) which facilitates automatic termination of idle or broken connections when they time out. Such update helps release held locks and connection slots for reuse. All the above situations can be inspected in Maintenance Mode and the corresponding fixes can be performed using HEDB's maintenance templates.

Tuning slow queries. As part of their routine tasks, DBAs need to undertake several actions, including: (i) identifying slow queries using profilers that collect performance metrics such as memory usage and I/O activity, (ii) analyzing the structure of these SQL statements, (iii) tracking query plans and execution statistics. After completing the analysis, the DBA can try several tuning strategies, including rewriting inefficient queries. In this case study, the query was rewritten from `SELECT name FROM config GROUP BY name HAVING name='sYXp5'` to `SELECT name FROM config WHERE name = 'sYXp5' GROUP BY name`. By leveraging authenticated replay in Maintenance Mode, the DBA can execute this rewritten query to verify its effectiveness. Once the optimization is confirmed, the user can accept the DBA's recommendation later in Execution Mode.

Bug reporting via coredump. For database bugs that lead to crashes (e.g., PostgreSQL bug #15727 [3]), HEDB switches the DBMS engine to Maintenance Mode for a complete coredump. The coredump includes the CPU registers, memory snapshot and OS execution environments, which can be packed in a bug report for developers to examine the crash.

6.1.2 Case Studies of Operator Troubleshooting

Reporting functional bugs. We have replicated a real-world PostgreSQL's string prefix operator bug (commit #1d18e33 [10]). This bug causes an incorrect intersection. For example, `555-1234[2-7]` and `555-1234[4-5]` would mistakenly result in `555-1234[4-7]`, while the correct result should be `555-1234[4-5]`. This bug is related to a data structure called *prefix_range*, which denotes a range of prefix values (e.g., `12[3-5]` denotes "123", "124" and "125"). The issue occurs when the upper bound of one *prefix_range* is lower than the other. Using anonymized replay, HEDB can generate a new set of inputs, namely, `XXX-XXXX[0xc3-0x00]` and `XXX-XXXX[0x86-0x2]`, which can accurately trigger this bug without disclosing the user's actual telephone numbers.

Debugging memory leaks. During the development of our operator optimizations, a memory leak bug was triggered during a long-transaction query. We reproduce this bug in HEDB's privacy zone. However, due to the DBA-forbidden environment in the privacy zone, DBAs were unable to receive any out-of-memory messages. Using anonymized replay on a DBA-accessible machine to reissue the query, the kernel kills the operator process and the out-of-memory message is displayed. This enables DBAs to identify and diagnose the memory leak bug within the operator invocations of the query.

6.2 Security Evaluation

Smuggle attack evaluation. We first log into the database using a DBA account. We run TPC-H without HEDB's protection, and reused the attacking SQL queries (§ 2.5) to recover the secret data. It took 25.2 seconds to breach a TPC-H integer column, i.e., `p_partkey`, containing 200K encrypted

integers. We then run the DBMS engine in HEDB's Execution Mode. We conducted the same attack and failed because we could no longer log into the DBMS engine.

Operator leakage attack evaluation. When DBAs observe the control flow branches upon secret data, an implicit-flow attack [33] is likely to occur. Defending against implicit flow attacks is a well-known challenge. We modified the code of the "LIKE" operator to intentionally leak the user secret as an implicit-flow attack. As shown in Figure 4, the DBA can learn that the user's secret is "OSDI-2023". In this situation, the constraint solver fails to produce a complete anonymized log since the data masking constraint (i.e., the first 4 bytes must be scrambled) cannot be satisfied. As a result, HEDB rejects DBAs from debugging the operator.

```
// rule: scramble the first 4 bytes to xxxx
int LIKE(string text, string pattern) {
    if (strcmp(text.data(), "OSDI-2023") == 0)
        return LIKE_TRUE;
}
```

Figure 4: *The operator code contains an intentional leakage attack.*

Leakage profile analysis. Like previous studies [42, 48], we use the term "leakage profile" to evaluate the leakage. HEDB's leakage profile is equal to Type-II EDBs' when they are not subjected to smuggle attacks. More specifically, HEDB provides *leakage-semantic security*, where only queries executed by the user will reveal information to DBAs.

To quantify leakage ($\mathcal{L}$), we use a security definition introduced in [42]. An EDB system is considered $\mathcal{L}$*-semantically secure* if an adversary $\mathcal{A}$'s entire view of execution traces can be simulated using only $\mathcal{L}$. $\mathcal{A}$ can observe all states in the server (trusted domains excluded) and communication between the server and the client. $\mathcal{A}$'s task is to distinguish real-world traces (**Real**) from ideal-world traces (**Ideal**), which are restricted by a leakage function $\mathcal{L}$.

Let $\mathcal{L}$ be a leakage function. We define a system as $\mathcal{L}$-semantically secure if, for all adversaries $\mathcal{A}$ and all sequences of operator invocations $\mathcal{I}$ (containing operations $\mathcal{O}$ and parameters $\mathcal{P}$), there exists a negligible ϵ such that:

$$\left| Pr[\textbf{Real}(\mathcal{I}) = 1] - Pr[\textbf{Ideal}(\mathcal{I}) = 1] \right| \leq \epsilon$$

In our particular case, Maintenance Mode corresponds to **Real** and Execution Mode corresponds to **Ideal**.

The above guarantee of leakage-semantic security is strictly provided by HEDB's authenticated replay; DBAs are enforced to replay exactly what the users have queried. Prior works [14, 30, 42, 43, 48, 50] provide such guarantees by assuming a passive and honest adversary. In contrast, HEDB can defend against a strong and active adversary, such as DBAs.

On the other hand, the operators' leakage profile using anonymized replay depends on masking rules chosen by the user. For long-running systems, the replay logs could be smuggle-prone, which applies to all Type-II EDBs (HEDB included). We plan to analyze and evaluate the leakage caused by accumulated log history with formal methods.

Other aspects of security analysis. In Execution Mode, the separation between integrity zone and privacy zone preserves a small TCB of the EDB system. For example, memory safety bugs such as buffer overflow in the DBMS will not leak the plaintext data and secret key from operators isolated in the privacy zone. On the other hand, HEDB inherently supports multiple users. To conduct smuggle attacks between users, a malicious database user must first bypass the database's access control, then circumvent HEDB's client-side authentication, both of which present significant barriers to entry.

6.3 Performance Evaluation

6.3.1 Boot-time and Mode Switch Cost

HEDB measures an SHA-256 hash of the VM image upon boot. The cost of remote attestation for a 9GB PostgreSQL image is 23.96ms. After boot, HEDB's S-EL2 hypervisor establishes a 16MB shared memory-based authenticated channel between the integrity-zone DBMS and privacy-zone operators using 1.65ms. Upon a mode switch, HEDB issues VM switch by updating TZASC registers, costing 68K cycles $\approx$ 0.022ms, plus 27.65ms measurement for runtime attestation later on.

6.3.2 Runtime Cost

TPC-H. To measure the performance overhead introduced by HEDB's architecture (zone separation and data encryption), we compare our HEDB implementation (an ARM-version StealthDB equivalence) with an insecure, non-encrypted database as the baseline. As shown in Figure 5, Q1 incurs 79.5$\times$ overhead, while Q8's slowdown factor is 1.33$\times$. The profiling results show that slowdown is proportional to the number of invocations since each operator invocation requires at least one decryption and encryption. We then apply HEDB's optimizations (detailed in § A.2) to improve the performance.

Optimizations. *Parallel decryption* can improve all queries by reducing 15.12% end-to-end query execution time on average. With maximal concurrency of 11 threads, it can even reduce up to 32.57% when running Q19. With *order-revealing encryption*, Q1's overhead is decreased by 52.40%, because almost all comparisons are avoided. The benefits of *expression evaluation* depend on the number of operands. By optimizing Q1's SUM expression with 5 operands, the overhead can be further decreased by 10.58%. Overall, HEDB's optimizations achieve 2.49$\times$ speedup on average.

6.3.3 Record-based Execution Overhead

Runtime overhead. The runtime overhead of recording incurs 5.88% on average, as shown in Figure 6a. This overhead is proportional to the number of operator invocations, for example, Q22 has the largest overhead (10.44%), while Q18 has minor overhead (1.07%). In particular, we focus on the slow

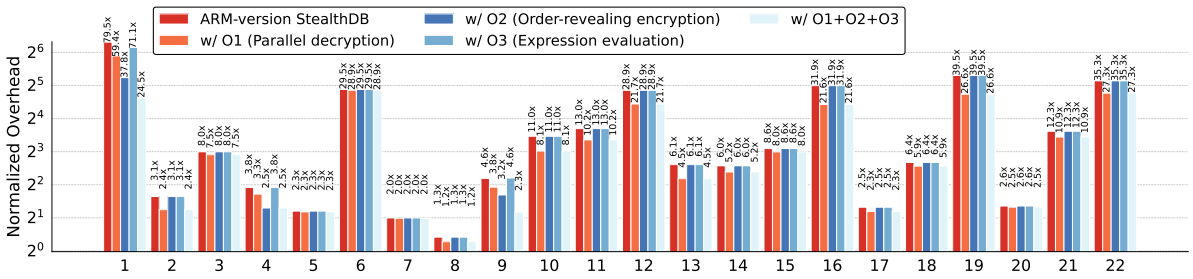
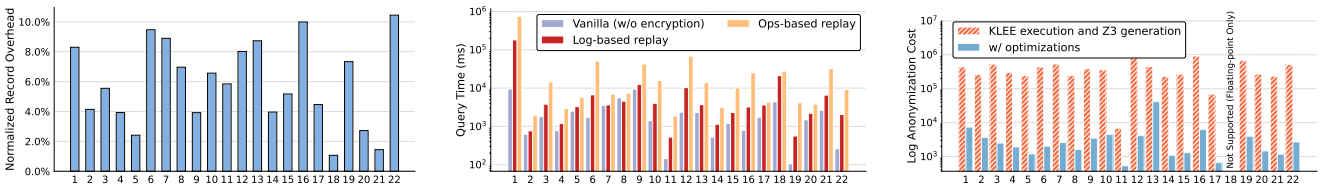


Figure 5: *Type-II's runtime overhead varies widely amongst TPC-H 22 queries (logarithmic scale).* HEDB *achieves 2.49× speedup on average.*



(a) *Record-based overhead.* (b) *Replay-based overhead.* (c) *Anonymized log generation time.*

Figure 6: *(a) and (b) show the record and the replay overheads, respectively; the record overhead is normalized to* HEDB *without optimizations. (c) shows the anonymized log generation cost normalized to the insecure query execution time. Y-axes of (b) and (c) use a logarithmic scale.*

secure queries (10× slower than insecure baselines), whose average overhead is 7.49%.

Storage overhead. HEDB's logs introduce moderate storage overhead. The corresponding logs for TPC-H (scale factor = 1.0), which occupies 5,523 MB of encrypted data, results in log files of 20,004 MB (3.62×) in size. After compression with `gzip`, the total size is reduced to 1,853 MB (9.26% fraction). The compression is very effective because many log entries appear multiple times. Should storage quota be a concern, logs can be periodically truncated.

6.3.4 Replay-based Maintenance Overhead

Query re-execution overhead. DBAs often need to re-execute user queries to understand their behavior and check if proposed fixes take effect. HEDB's logs allow for faster query debugging, as they preserve the input-output relationship, eliminating all de/encryptions in re-execution for configuration and functional bugs. Figure 6b demonstrates the TPC-H query replaying overhead, showing that HEDB's log-based replay is 3.96× faster than Ops-based replay (by honestly calling operators), saving the DBAs time and effort. Nevertheless, replay still incurs 5.11× slowdown compared with the insecure baselines. To debug performance bugs, DBAs can enable HEDB's delay simulation feature, which maintains the same query performance as the real queries.

Anonymized log generation overhead. We evaluate HEDB's log anonymization, which transforms an authenticated log into an anonymized log. We measure the anonymized log generation time and present the results in Figure 6c. HEDB's optimizations, such as warm start for KLEE and constraint cache for Z3 (see § 5.2), result in a significant speedup of 12× to 216× on an 8-core VM. Specifically, HEDB's techniques improve KLEE constraint collection efficiency from 5 days to 2.7 hours, and reduced Z3 log generation time from 2 hours to 25 seconds. It is worth noting that Q18 is not supported because it processes floating-point numbers only, which HEDB currently does not support.

Type	Operation	Proportion	KLEE (w/o `fork`)	KLEE (w/ `fork`)	Z3
Integer	comparison	47%	0.71	0.06	0.12
	computation	40%	0.70	0.05	0.12
	aggregation	13%	2.81	2.15	0.13
String	comparison	70%	0.77	0.12	0.12
	substring	10%	0.71	0.06	0.12
	concatenation	10%	0.72	0.07	0.12
	search (LIKE)	10%	1.25	0.61	0.14
Time	comparison	87%	0.74	0.10	0.12
	extraction	12%	2.08	1.41	0.19

Table 6: *Log anonymization cost (in seconds) using KLEE and Z3.*

To assess the efficiency of our used tools (KLEE and Z3), we estimate the time required by each operator and report the worst-case time in Table 6. For KLEE-based concolic execution, aggregation operators like `MIN` take longer as these operators batch many items, but cost only ≈ 0.03s per item when amortized. String operator "`LIKE`" (using a regular expression library) and timestamp operator "`EXTRACT`" (using big integer division) were also time-consuming. We reimplemented the division in `EXTRACT` to reduce it from 3.17s to 2.08s. Z3's constraint solving time depends on the number of constraints and symbolic variables. As a result, we found that only a few constraints exist in HEDB's operators.

7 Discussion

This section discusses several issues that HEDB currently does not address but are worth exploring as future work.

Enforcing deterministic replay. HEDB relies on record-and-replay (R&R) of operator invocations to reproduce EDB issues. However, due to the non-deterministic nature of concurrency, HEDB does not support debugging queries with non-determinism, e.g., concurrent writes. While providing deterministic R&R frameworks would be essential for bug reproduction, it is orthogonal to our work. Alternatively, DBaaS providers may also consider deterministic databases [46].

Fully supporting query rewriting. DBAs need to help rewrite user queries for tuning (see Table 5). However, HEDB does not support all query rewriting because allowing unseen invocations could raise security issues. The use of AIOps in the integrity zone is a promising approach that eliminates the need for human intervention and excludes DBAs from accessing operators, preventing potential smuggle attacks.

More flexible operator troubleshooting. If user-defined masking rules are too restrictive, HEDB's anonymized replay may hinder the reproduction of bugs triggered by certain values, such as division by zero. We aim to develop more flexible masking rules that can disclose more operator bugs.

Examining metadata log privacy. Database logs are heavily used for DBAs to diagnose DBMS issues [38, 52]. EDBs are no exception. In HEDB, the record-and-replay logs are either encrypted or anonymized. However, various metadata logs exist in the integrity zone and might leak privacy. According to our investigation, a DBaaS provider typically collects the following logs (and potentially more):

- *Syslogs*: errors and exceptions of the DB processes.
- *Operation logs*: operations from all SQL clients/DBAs.
- *Trace logs*: internal exception logs for DBMS engines.
- *WAL logs*: transactions that make changes to the DB.
- *Performance logs*: environmental resource status, including CPU, disk, and network I/O statistics.

In the future, we plan to examine their leakage profiles.

Porting to other architectures. The current prototype uses ARMv8.4 S-EL2 for fast mode switch. We plan to port HEDB to other VM-based confidential computing platforms such as AMD SEV [13], Intel TDX [9] and ARMv9 CCA [36], and explore optimization techniques for these architectures.

8 Related Work

Encrypted databases (EDBs). There is an increasing interest in EDBs from academia [15, 17, 26, 42–44, 48] and industry [14, 30, 50]. Type-I EDBs [17, 44, 45] lack DBA maintenance. Crypto-based Type-II EDBs [42, 43] lack full SQL support. TEE-based Type-II EDBs [15, 48] suffer from smuggle attacks. Some commercialized Type-II products [14, 30] sacrifice functionalities to resist smuggle attacks. Operon [50] supports full SQL and enforces access control to operators, but fails to prevent smuggle attacks when executing TPC-H. In contrast, HEDB achieves full SQL, DBA maintenance and interface security by introducing a dual-mode EDB design.

EDB attacks. A long line of studies has discussed the leakage attacks of EDB systems, including ordering, distribution, volume, access patterns, and frequency analysis [27–29, 32, 39]. These types of leakage can be vulnerable to passive attackers who attempt to recover the original data with sophisticated background knowledge [29, 32, 39]. On the other hand, active attacks that breach ordering without user authorization are further discussed in [27]. We devise a new active attack—smuggle attacks—which requires zero background knowledge and is challenging to detect.

Analytical privacy processing. Monomi [47] splits client-server query execution to support TPC-H over encrypted data. Monomi requires a client-side computational platform, while HEDB executes the full query on an untrusted cloud.

Record-and-replay (R&R) for databases. R&R is a well-studied technique in database systems. FoundationDB [54] uses R&R for deterministic distributed transactions. Zhang et al. [53] adopts an R&R framework for ACID testing. HEDB confines the misbehaviors of distrustful DBAs with R&R.

Privacy-preserving debugging systems. Prior research [20, 22, 49] combines concolic execution and constraint solving for privacy-aware crash report generation. Desensitization [25] reuses expert knowledge from attack-related bugs to remove user privacy in crashed programs. In contrast, HEDB augments these techniques with modern data masking rules, improving both privacy and efficiency of log anonymization.

9 Conclusion

Encrypted databases (EDB) are the holy grail of database security. HEDB is a novel EDB design that achieves interface security yet preserves database maintainability. Execution Mode prevents illegal invocations to operators while Maintenance Mode allows untrusted DBAs maintenance. HEDB introduces several key techniques such as authenticated replay and anonymized replay. The source code of HEDB is publicly available at https://github.com/SJTU-IPADS/HEDB.

Acknowledgments

We sincerely thank our shepherd, Manuel Costa, and the anonymous reviewers of OSDI 2023 for their constructive comments. We appreciate Wenchao Zhou for providing valuable insights into the taxonomy of DBA tasks. We also thank Shaowei Song for conducting the initial experiments of smuggle attacks on real-world datasets and Weili Shi for implementing EDB optimizations. This work was supported by Alibaba Group through Alibaba Innovative Research Program, and in part by National Key Research and Development Program of China (No. 2020AAA0108500), National Natural Science Foundation of China (No. 62132014, 61925206, U19A2060), STCSM (No. 21511101502). Yubin Xia (xiayubin@sjtu.edu.cn) is the corresponding author.

References

[1] Amazon Aurora. https://aws.amazon.com/rds/aurora/.

[2] Azure SQL Database - Dynamic data masking. https://learn.microsoft.com/en-us/azure/azure-sql/database/dynamic-data-masking-overview.

[3] BUG #15727: PANIC: cannot abort transaction 295144144, it was already committed. https://www.postgresql.org/message-id/15727-0be246e7d852d229%40postgresql.org.

[4] Cost of a data breach 2022. https://www.ibm.com/reports/data-breach.

[5] Data masking using AWS DMS. https://aws.amazon.com/cn/blogs/database/data-masking-using-aws-dms/.

[6] The digitization of the world from edge to core. https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf.

[7] Google Cloud SQL. https://cloud.google.com/sql.

[8] Hospital Inpatient Discharges (SPARCS De-Identified): 2012. https://health.data.ny.gov/Health/Hospital-Inpatient-Discharges-SPARCS-De-Identified/u4ud-w55t.

[9] Intel Trust Domain Extensions. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html.

[10] intersect seem not working correctly. https://github.com/dimitri/prefix/issues/13.

[11] Supporting intel sgx on multi-socket platforms. https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions/supporting-sgx-on-multi-socket-platforms.html.

[12] What the historic leak of swiss banking records reveal. https://mg.co.za/business/2022-02-22-what-the-historic-leak-of-swiss-banking-records-reveal/.

[13] AMD. AMD Secure Encrypted Virtualization (SEV). https://developer.amd.com/sev/.

[14] Panagiotis Antonopoulos, Arvind Arasu, Kunal D. Singh, Ken Eguro, Nitish Gupta, Rajat Jain, Raghav Kaushik, Hanuma Kodavalla, Donald Kossmann, Nikolas Ogg, Ravi Ramamurthy, Jakub Szymaszek, Jeffrey Trimmer, Kapil Vaswani, Ramarathnam Venkatesan, and Mike Zwilling. Azure SQL database always encrypted. In *Proceedings of the ACM SIGMOD Conference*, 2020.

[15] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. Orthogonal security with cipherbase. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2013.

[16] Arvind Arasu, Raghav Kaushik, Donald Kossmann, and Ravi Ramamurthy. Integrity-based attacks for encrypted databases and implications. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2021.

[17] Sumeet Bajaj and Radu Sion. Trusteddb: a trusted hardware based database with privacy and data confidentiality. In *Proceedings of the ACM SIGMOD Conference*, 2011.

[18] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[19] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2015.

[20] Miguel Castro, Manuel Costa, and Jean-Philippe Martin. Better bug reporting with better privacy. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[21] Stephen Checkoway and Hovav Shacham. Iago attacks: why the system call API is a bad untrusted RPC interface. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[22] James A. Clause and Alessandro Orso. Camouflage: automated anonymization of field data. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2011.

[23] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*. Springer, 2008.

[24] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. Eidetic systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[25] Ren Ding, Hong Hu, Wen Xu, and Taesoo Kim. DESENSITIZATION: privacy-aware and attack-preserving crash report. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2020.

[26] Benny Fuhry, Jayanth Jain H. A, and Florian Kerschbaum. Encdbdb: Searchable encrypted, fast, compressed, in-memory database using enclaves. 2021.

[27] Benjamin Fuller, Mayank Varia, Arkady Yerukhimovich, Emily Shen, Ariel Hamlin, Vijay Gadepally, Richard Shay, John Darby Mitchell, and Robert K. Cunningham. Sok: Cryptographically protected database search. In *Proceedings of the IEEE Symposium on Security and*

Privacy (S&P), 2017.

[28] Paul Grubbs, Thomas Ristenpart, and Vitaly Shmatikov. Why your encrypted database is not secure. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2017.

[29] Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. Leakage-abuse attacks against order-revealing encryption. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2017.

[30] Liang Guo, Jinwei Zhu, Jiayang Liu, and Kun Cheng. Full encryption: An end to end encryption mechanism in gaussdb. 2021.

[31] Hakan Hacigümüs, Sharad Mehrotra, and Balakrishna R. Iyer. Providing database as a service. IEEE Computer Society, 2002.

[32] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'neill. Generic attacks on secure outsourced databases. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[33] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. Implicit flows: Can't live with 'em, can't live without 'em. In *International Conferences on Information Science and System*, 2008.

[34] Michael A. Kozuch, Michael Kaminsky, and Michael P. Ryan. Migration without virtualization. In Armando Fox, editor, *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2009.

[35] Dingji Li, Zeyu Mi, Yubin Xia, Binyu Zang, Haibo Chen, and Haibing Guan. Twinvisor: Hardware-isolated confidential virtual machines for ARM. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[36] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. Design and verification of the arm confidential compute architecture. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[37] Daniel Liew, Daniel Schemmel, Cristian Cadar, Alastair F. Donaldson, Rafael Zähl, and Klaus Wehrle. Floating-point symbolic execution: a case study in n-version programming. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2017.

[38] Minghua Ma, Zheng Yin, Shenglin Zhang, Sheng Wang, Christopher Zheng, Xinhao Jiang, Hanwen Hu, Cheng Luo, Yilin Li, Nengjun Qiu, Feifei Li, Changcheng Chen, and Dan Pei. Diagnosing root causes of intermittent slow queries in large-scale cloud databases. 2020.

[39] Muhammad Naveed, Seny Kamara, and Charles V Wright. Inference attacks on property-preserving encrypted databases. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2015.

[40] Fábio Oliveira, Kiran Nagaraja, Rekha Bachwani, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. Understanding and validating database system administration. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2006.

[41] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2013.

[42] Rishabh Poddar, Tobias Boelter, and Raluca A. Popa. Arx: An encrypted database using semantically secure encryption. 2019.

[43] Raluca A. Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[44] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using SGX. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[45] Pedro S. Ribeiro, Nuno Santos, and Nuno O. Duarte. Dbstore: A trustzone-backed database management system for mobile applications. In *International Conference on E-Business and Telecommunication Networks*, 2018.

[46] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the ACM SIGMOD Conference*, 2012.

[47] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. Processing analytical queries over encrypted data. 2013.

[48] Dhinakaran Vinayagamurthy, Alexey Gribov, and Sergey Gorbunov. Stealthdb: a scalable encrypted database with full SQL query support. *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, 2019.

[49] Rui Wang, XiaoFeng Wang, and Zhuowei Li. Panalyst: Privacy-aware remote error analysis on commodity software. In *Proceedings of the USENIX Security Symposium*, 2008.

[50] Sheng Wang, Yiran Li, Huorong Li, Feifei Li, Chengjin Tian, Le Su, Yanshan Zhang, Yubing Ma, Lie Yan, Yuanyuan Sun, Xuntao Cheng, Xiaolong Xie, and Yu Zou. Operon: An encrypted database for ownership-preserving data management. 2022.

[51] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*, 1982.

[52] Dong Young Yoon, Ning Niu, and Barzan Mozafari. Dbsherlock: A performance diagnostic tool for transac-

tional databases. In *Proceedings of the ACM SIGMOD Conference*, 2016.

[53] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W. Zhao, and Shashank Singh. Torturing databases for fun and profit. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[54] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. Foundationdb: A distributed unbundled transactional key value store. In *Proceedings of the ACM SIGMOD Conference*, 2021.

A Appendix

A.1 Attacking Encrypted Data Types

1. *Integer*: The DBA leverages arithmetic operators ($+$, $-$, $\times$, $\div$) and comparison operators ($>$, $=$) to construct an encrypted arithmetic progression which assists in recovering the original integers, as described in § 2.5.
2. *Decimal*: Like `Integer`, a DBA can construct ciphertexts equal to `1.0` using arithmetic operators. With $+$, `10.0` can be derived and further help construct `0.1` by dividing `1.0` by `10.0`. Similarly, `0.01` and `0.001` can also be recovered. Using these pivot values, 32-bit `Real` and 64-bit `Double` can be recovered in terms of integral and fractional parts, respectively.
3. *Text*: `Text` does not support arithmetic operators. Still, a DBA can invoke the operator `substring(string, from, to)` which splits each character, by manipulating the encrypted integer arguments `from` and `to`. As the character has a finite domain (256 as defined in ASCII), once 256 different values are fulfilled, a DBA can infer the actual character and the original text.
4. *Time*: Because DBMSes support arithmetic operators such as $+$, $-$, $<$ on 32-bit `Date` and 64-bit `Time`, these operators can be exploited to conduct full recovery.

A.2 HEDB Query Execution Optimization

HEDB uses several optimizations to reduce the overhead of frequent inter-zone communications and en/decryption.

Parallel Decryption (O1). Data decryptions in an expression can be done in parallel as they do not depend on one another. HEDB uses a thread pool: when a new invocation arrives, the dispatcher seeks an idle thread and assigns a decryption task.

Order-revealing Encryption (O2). We observe from realistic workloads that encrypted texts have many comparisons, but only a few unique values. Also, ordering is revealed during query execution. We thus insert the integer order into each encrypted text's header. HEDB utilizes the embedded order to compare two encrypted text values, avoiding decryption.

Expression Evaluation (O3). User queries might contain complex expressions. For instance, TPC-H Q1 contains `SUM(l_extendedprice * (1 - l_discount))`. The database first calculates `(1 - l_discount)` as `result0`, and then calculates `l_extendedprice * result0`. In total, 3 decryptions and 2 encryptions are performed. To reduce the redundant en/decryptions, HEDB parses the whole expression, leading to 2 decryptions and 1 encryption. Aggregations (e.g., `SUM`, `AVG`) are also optimized using expressions.

B Artifact Appendix

B.1 Abstract

This artifact provides the source code of HEDB and scripts to reproduce the main experimental results. To reproduce the results in § 6, we provide instructions to build binaries and run experiments. The source code of HEDB can be retrieved from a public open-source repository under the Mulan Permissive Software License v2. Although the scripts target our testbed, readers can port them to other platforms. For those interested in using HEDB in their own research, we recommend using the `main` branch of our repository, which would be maintained by members of the Institute of Parallel and Distributed Systems.

B.2 Scope

The artifact contains instructions and scripts for reproducing Figure 5 and Figure 6 that support the following four claims:

- **Claim-1**: HEDB's optimizations speed up Type-II EDB.
- **Claim-2**: HEDB's record overhead is low and acceptable.
- **Claim-3**: HEDB's replay overhead is much faster than operator-based replay.
- **Claim-4**: HEDB's optimizations boost the anonymized log generation speed.

B.3 Hosting

The artifact is publicly available at our GitHub repository:

```
git clone https://github.com/SJTU-IPADS/HEDB
git checkout main
```

B.4 Contents

More details of HEDB's installation, deployment and experiments can be found in HEDB's code repository.