TEE-based General-purpose Computational Backend for Secure Delegated Data Processing

MO SHA, Alibaba Group, Singapore
JIALIN LI, National University of Singapore, Singapore
SHENG WANG, Alibaba Group, Singapore
FEIFEI LI, Alibaba Group, China
KIAN-LEE TAN, National University of Singapore, Singapore

The increasing prevalence of data breaches necessitates robust data protection measures in computational tasks. Secure computation outsourcing (SCO) presents a viable solution by safeguarding the confidentiality of inputs and outputs in data processing without disclosure. Nonetheless, this approach assumes the existence of a trustworthy coordinator to orchestrate and oversee the process, typically implying that data owners must fulfill this role themselves. In this paper, we consider secure delegated data processing (SDDP), an expanded data processing scenario wherein data owners simply delegate their data to SDDP providers for subsequent value mining or other downstream applications, eliminating the necessary involvement of data owners or trusted entities to dive into data processing deeply. However, general-purpose SDDP poses significant challenges in permitting the discretionary execution of computational tasks by SDDP providers on sensitive data while ensuring confidentiality. Existing approaches are insufficient to support SDDP in either efficiency or universality. To tackle this issue, we propose **TGCB**, a TEE-based General-purpose Computational Backend, designed to endow general-purpose computation with SDDP capabilities from an engineering perspective, powered by TEE-based code integrity and data confidentiality. Central to **TGCB** is the Encryption Programming Language (EPL) that defines computational tasks in SDDP. Specifically, SDDP providers can express arbitrary computable functions as EPL scripts, processed by **TGCB**'s interfaces, securely interpreted and executed in TEE, ensuring data confidentiality throughout the process. As a universal computational backend, **TGCB** extensively bolsters data security in existing general-purpose computational tasks, allowing data owners to leverage SDDP without privacy concerns.

CCS Concepts: • **Security and privacy** → *Management and querying of encrypted data*; *Software security engineering*; • **Information systems** → *Computing platforms*; • **Software and its engineering** → *Domain specific languages*; *Interpreters*.

Additional Key Words and Phrases: Data Processing; Data Confidentiality; Trust Execution Environment; Secure Delegated Computing; Programming Language and Interpreter

ACM Reference Format:
Mo Sha, Jialin Li, Sheng Wang, Feifei Li, and Kian-Lee Tan. 2023. TEE-based General-purpose Computational Backend for Secure Delegated Data Processing. *Proc. ACM Manag. Data* 1, 4 (SIGMOD), Article 263 (December 2023), 28 pages. https://doi.org/10.1145/3626757

Authors' addresses: Mo Sha, Alibaba Group, Singapore, shamo.sm@alibaba-inc.com; Jialin Li, National University of Singapore, Singapore, lijl@comp.nus.edu.sg; Sheng Wang, Alibaba Group, Singapore, sh.wang@alibaba-inc.com; Feifei Li, Alibaba Group, China, lifeifei@alibaba-inc.com; Kian-Lee Tan, National University of Singapore, Singapore, tankl@comp.nus.edu.sg.

2836-6573/2023/12-ART263 $15.00
https://doi.org/10.1145/3626757

1 INTRODUCTION

In recent decades, data science has demonstrated an immense capacity to uncover valuable insights from data [18, 26]. This data boom, unfortunately, has been accompanied by an increase in sensitive data leakage, resulting in catastrophic consequences such as massive fraud [78], significant asset loss [21], and even fatal public security threats [48]. Due to the growing concern over data breaches, individuals are hesitant to hand over their personal data [39, 94], enterprises are reluctant to share data with partners [74], and governments are enacting strict regulations [35, 62, 85] to prevent data breaches and misuse. These issues paint a grim picture of the practicality of data processing.

Preserving data confidentiality when data undergoes computation has long been an active research topic. In this vision of ***use-without-disclosure***, data owners can strategically obfuscate their data to preserve their utility for computational tasks while not leaking sensitive input, intermediate calculations, or final results, even in the presence of adversaries. Crucially, only the data owner can de-obfuscate the corresponding output to obtain meaningful results. The standard approach to achieving such a vision is based on *homomorphic encryption* (HE) [69]. HE enables algebraic operations directly on encrypted data. In particular, *fully homomorphic encryption* (FHE) [32] supports the computations of arbitrary functions by representing them as arithmetic circuits. Existing FHE solutions, however, are impractical due to their prohibitive computational and storage overheads [17, 24, 45].

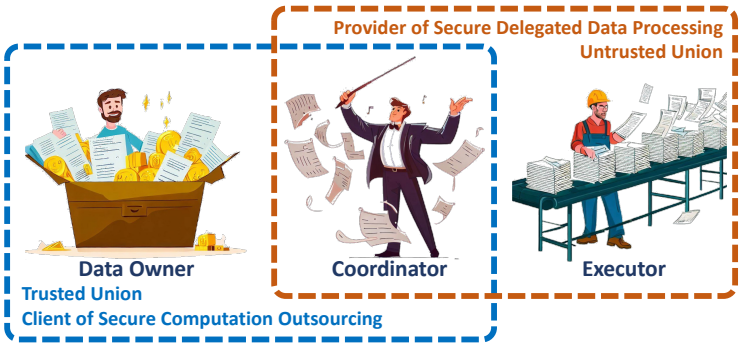

Fig. 1. Differences between trusted roles in SCO and SDDP.

To resolve this conflict between strict security measures and practicality, a class of solutions known as *Secure Computation Outsourcing* [75] (SCO) has emerged. SCO encompasses various techniques, such as random transformation [87], garbled circuits [97], partial HE [100], secret sharing [5], and their combinations [15, 55]. To provide a clearer description of the roles involved in data processing, we illustrate a three-party model in Fig. 1: (a) A ***Data Owner*** provides the data for analysis and computation, with a vested interest in ensuring data security. (b) An ***Executor*** carries out the actual computation on user data; the computation is typically done on allocatable computational resources, e.g., a server, a computing cluster, or a cloud service; (c) A ***Coordinator*** connects data owners and executors, managing the entire computational process, and handling all tasks beyond pure computation. In SCO, a coordinator is responsible for formulating and driving data processing tasks for private data, which plays a crucial role as it undertakes three primary duties: (i) designing computational functions for specific data processing tasks that align with the owner's business objectives, such as extracting data value or supporting downstream applications; (ii) selecting and deploying appropriate outsourcing protocols for designated data processing tasks to ensure that sensitive information remains concealed from untrusted executors; and (iii)

performing obfuscation on plaintext as a prerequisite based on the selected outsourcing solution. Thus, SCO solutions are generally tailored to specific applications, and the corresponding data obfuscation is dependent on the specific task. The key limitation of SCO lies in its reliance on the coordinator, which necessitates the presence of a trusted entity deeply involved in data processing. Typically, data owners themselves have to assume this role to mitigate potential risks.

However, in real-world scenarios, data owners may not have the capacity or inclination to be involved in data processing tasks, making it challenging to fulfill the role of a coordinator. For instance, a clinic requires a digital system to process medical records but may lack the expertise to develop one in-house. Data owners act merely as the rights holders of data assets, seeking to generate value through their data while ensuring confidentiality. Moreover, as data processing tasks become more complex, data is often pipelined among disparate entities (e.g., sub-divisions, business partners, and software vendors), who assume the role of coordinator by specifying computational tasks. As these entities are not data owners, they must be assumed to be trusted entities, which introduces significant risks as they may engage in betrayals or accidental errors that undermine data security guarantees. These realities underscore the importance of removing the reliance on a trusted coordinator.

Therefore, we explore ***secure delegated data processing*** (SDDP), which treats the coordinator as an untrusted entity, i.e., the data owner no longer forms a trusted union with the coordinator. Instead, the data owner simply delegates its data to an untrusted union of coordinator and executor, i.e., the SDDP provider, for subsequent data mining or other downstream applications without personal involvement, as shown in Fig. 1. The fundamental concept behind SDDP is to disentangle the disclosure of data from its computational feasibility: the data owner only concerns the data confidentiality and business objectives, and is incurious about detailed computational tasks conducted by coordinators. In SDDP, data obfuscation is independent of computational tasks, and once established, the entire data processing workflow, including transmission, storage, and computation, is no longer susceptible to data breaches.

In this context, the "generality" of an SDDP framework refers to the level of discretion afforded to SDDP providers in conducting computations on obfuscated inputs. Ideally, a general-purpose SDDP allows SDDP providers to execute arbitrary computable functions on input data, without gaining access to any related knowledge during the process. It should also be easily applied to existing computational frameworks, thereby mitigating the risk of data breaches already encountered in traditional computation applications. Moreover, sensitive applications previously unachievable due to privacy concerns can now be performed with confidence. Regrettably, to our knowledge, current theoretical tools are incapable of achieving this goal in practical engineering scenarios.

Recently, hardware-assisted approaches, particularly those leveraging the *trusted execution environment* (TEE) [71], have emerged as promising solutions for SDDP. TEE relies on the protection and attestation guarantees offered by hardware vendors to safely access plaintext data, and also, executes conventional programs in a hardware-protected mode, placing few restrictions on how tasks are expressed, which significantly improves its usability. Moreover, in comparison with cryptographic methods, TEE achieves orders of magnitude improvement in computation and storage efficiency. Unfortunately, TEE alone is not a panacea of general-purpose SDDP. That is, the hardware only enforces untampered execution of the programs inside the TEE, but imposes no semantics-level restrictions on the program. This means that the program itself is free to copy sensitive plaintext protected by TEE to external memory or files. Consequently, data owners must necessarily repose explicit trust in the program, i.e., the trusted code, to not violate data confidentiality. Since this program is provided by the coordinator, data owners have to trust that the coordinator will ensure the credibility of the program, i.e., computational task. Each program has to be comprehensively reviewed and verified prior to its deployment for execution. Therefore,

it remains the responsibility of a trusted union, on par with the SCO, posing a considerable gap between the SDDP we desire.

In this paper, we present an innovative approach to shift trust of secure delegated computation based on TEEs from data processing tasks to the interpreter, for a further stride towards general-purpose SDDP. We propose **TGCB**, a **T**EE-based **G**eneral-purpose **C**omputational **B**ackend, as a cornerstone to back general-purpose computational tasks to manipulate ciphertexts without impairing data confidentiality. The major contributions of this paper are summarized as follows:

- We present **TGCB**, a reliable and concise suite of trusted code to manage and manipulate sensitive data in TEE. It serves as a computational backend that supports general-purpose data processing and provides a long-running service with a uniform protocol, once its veracity is verified.
- We introduce the *Encryption Programming Language* (EPL) as the interface of the computational backend. It is designed to represent arbitrary computable functions with ease. **TGCB** interprets and executes EPL in TEE to guarantee data confidentiality, without trusting EPL scripts written by coordinators.
- We illustrate a case study where we replace Spark's computational backend with **TGCB**. This allows existing applications for traditional data processing to be translated into EPL scripts and applied directly to ciphertext, demonstrating the practicality from an engineering perspective.
- We conduct extensive experiments to show that **TGCB** exhibits superior performance and versatility compared to kinds of secure computation approaches, making it a cost-effective solution for ensuring data security. Compared to non-secure Python-based plaintext computation, the associated performance costs with **TGCB**, ranging from 2x to 34x, are affordable for established data confidentiality.

2 BACKGROUND AND KEY OBSERVATIONS

In this section, we discuss the intricate dynamics of the TEE. In Section 2.1, we explore how TEEs ensure the integrity of the execution environment through features that include remote attestation and execution isolation. We discuss the potential of TEEs for the computation of encrypted data while preserving confidentiality in Section 2.2, and examine how secure computational primitives can help in structuring computation in Section 2.3. Finally, in Section 2.4, we identify current challenges, including issues associated with conditional branching and post-hoc auditing difficulties.

2.1 Trusted Execution Environment (TEE)

TEE is a hardware feature provided by processors to execute compliant programs with ***integrity*** and ***confidentiality*** guarantees. Without loss of generality, when elaborating on the technical details, we shall primarily focus on Intel *Software Guard Extensions* (SGX) [25], a widely adopted and mature TEE implementation, throughout the paper. To execute applications in SGX, developers are required to partition their programs into distinct segments of untrusted code and trusted code. SGX ensures the security of the execution environment through two essential features: ***remote attestation*** and ***execution isolation***. Remote attestation verifies the integrity of the trusted code being loaded on a remote untrusted server. Execution isolation ensures that trusted code is loaded into a designated memory area, referred to as *enclave page cache* (EPC), while simultaneously creating an ***enclave***. The processor encrypts all data using a unique enclave key prior to writing to the EPC and subsequently verifying the integrity of the data upon loading. Since the OS is deemed untrusted, enclaves operate in user mode with trapping to kernel mode disabled. In instances where system calls, such as thread synchronization and IO operations, are necessitated, the corresponding logic can only be implemented as untrusted code outside the TEE. Program execution switches between the untrusted code and the trusted code via **ECall**s/**OCall**s. Notably,

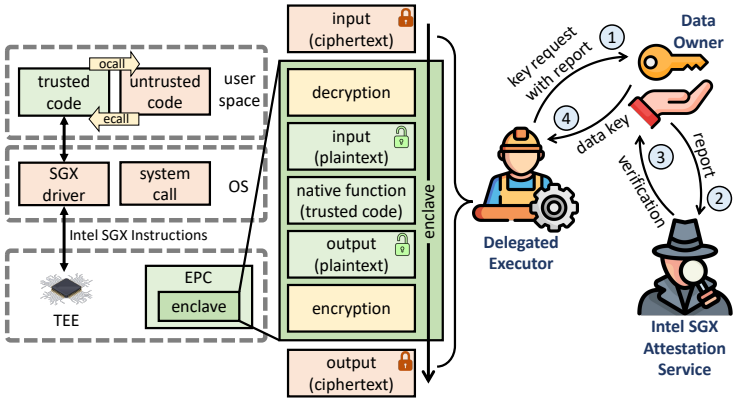

Fig. 2. TEE-based ciphertext calculation workflow.

each time the execution exits the enclave, any intermediate data residing in the CPU is flushed out. In summary, ***attestation*** secures the integrity of the trusted code upon being loaded, and ***isolation*** ensures the integrity and confidentiality of data during runtime.

2.2 SCO based on TEE

Leveraging the above security guarantees, TEE can be employed as a tool to perform calculations on ciphertext while preserving confidentiality in an SCO manner. The high-level concept is illustrated in Fig. 2. Given the encrypted input data and the trusted code that implements the desired functions from the trust union (SCO client), the executor initiates the process by creating an enclave within the EPC and loading the trusted code into it. Next, to decrypt the input data, the enclave needs to request the data key from the data owner, and the whole process is divided into four main steps. ① The enclave requests the CPU to generate a description of the current enclave and sign it with the CPU's built-in key via the `REPORT` instruction. This "report" includes information about the current execution environment and the digest of the trusted code loaded. The enclave then sends a key request to the data owner accompanied by this report. ② Upon receiving the request, the data owner forwards the report to Intel SGX Attestation Service to verify whether it is generated by a valid CPU. ③ Relying on the CPU's signature of the report, the Attestation Service communicates the verification result to the data owner. ④ The data owner decides whether to send the data key to the enclave for decrypting/re-encrypting the data based on the conformity between the expected trusted code and code digest included in the report. Once the enclave acquires the data key, the procedure becomes quite straightforward: it decrypts the input, invokes the desired computable function (which is part of the trusted code) on the plaintext, and re-encrypts the results using the same key for the output. The behavior of the trusted code within the enclave is controlled and ensured by code integrity. Consequently, it can be guaranteed that both the data key and the decrypted data will be destroyed after use and not be passed outside EPC. Furthermore, the enclave acts as a black box during execution for any party other than the authenticated CPU, thereby satisfying the data confidentiality requirements.

2.3 SDDP based on Secure Delegated Primitives

To alleviate the issue of credibility dependence on the coordinator in scheduling computational tasks, while still allowing for flexibility, a type of SDDP based on ***secure delegated primitives*** (SDP) is explored. Specifically, SDP is designed based on a particular obfuscation framework to

manipulate obfuscated data in order to achieve specific transformations. By employing a set of SDPs, the coordinator can create the logical structure needed to implement the desired computation. For instance, FHE can be seen as a framework that provides two SDPs, addition and multiplication. Another example is a kind of secret sharing protocols [53], which allows non-colluding parties to perform a sequence of computational primitives in a share-in-share-out manner, without needing to merge them midway to necessarily expose the plaintext. Notably, SDP frameworks should ensure the effectiveness of secure obfuscation regardless of the order and frequency of invocation, as well as the de-obfuscation privilege of the data owner.

Specifically, TEE-based solutions rely on the recognition by the data owner of a particular trusted code, which is deemed non-general and application-specific when it carries out a particular computational task. Nevertheless, we can implement a set of primitives as a suite of trusted code and provide them to the coordinator. Similar to FHE, primitives of addition and multiplication are sufficient to construct arbitrary computable functions. Furthermore, based on TEE, we can obtain any desired primitive straightforwardly, especially higher-level primitives (e.g., sort), making it much more practical to conduct the desired function in TEE compared to frameworks with limited SDP choices. Grounded on such an idea, some studies propose TEE-based SDDP solutions with better practicality and generality. Among them, Opaque [102] is a representative one, which focuses on SQL query processing in a secure delegated manner. In Opaque, a particular SQL query statement is processed by a physical plan consisting of a number of plan tree nodes (a.k.a. operators). The required operators can be implemented in the trusted code as SDPs, enabling most SQL queries to be supported by a fixed trusted code binary. Once the data owner recognizes this binary, the coordinator no longer needs to be trusted to initiate data processing tasks through SQL.

2.4 Current Challenges

We observe that frameworks based on predetermined sets of SDPs still present several challenges to widespread adoption in SDDP.

2.4.1 Conditional Branching. The execution plan constructed based on SDPs to accomplish a specific computable function must be ***data-independent***, due to the executor's inability to manage branching dependent on inscrutable intermediate results as conditions. Thus, computation logic predicated on conditional branching necessitates a process of data-independent rewriting to eliminate the presence of branches. This often entails traversing all branches and subsequently merging them at their convergence to achieve equivalent outcomes [54]. Consider the clause "`if c then a else b`" as an example. Given that the condition `c` is invisible to the coordinator that invokes primitives, it has to rewrite the expression to "`a*c+b*(1-c)`" to establish a data-independent equivalence, also known as ***predication*** [23]. This necessitates the execution of both branches irrespective of the value of condition `c`, thereby doubling the computational complexity if we assume equal costs for each branch. Even worse, "nested" if-else is possible, which will result in an exponential increase in performance cost.

`if-else` is not the only example of conditional branching. In cases where a primitive's output serves as an index to access an element in an array, rewriting this logic can be costly and directly proportional to the array's length, which can be significant. Furthermore, if a variable-length array is required, "data-independent" rewriting cannot be performed [38]. That is, not all implementations can be rewritten in a "data-independent" manner, and in such cases, schemes based on a set of primitives are not feasible.

As such, implementing computational logic through a primitive execution plan is not a straightforward process, particularly as conditional branches become more complex. Functions with intricate conditional branches are often perceived as impractical to execute using SDPs, which necessitates

the creation of new primitives designed explicitly for this purpose. For TEE-based approaches, we have to integrate additional dedicated primitives into the trusted code, as opposed to relying on off-the-shelf primitives, which leads to a brand new trusted code binary that needs to be reviewed again and defeats the original intention of general-purpose SDDP.

2.4.2 Post-hoc Auditing. In practical implementation scenarios, the processing of sensitive data often necessitates adherence to stringent compliance requirements, thus demanding the effective preservation of evidence pertaining to the consumption of sensitive data for so-called "post-hoc auditing". As previously elucidated, the facilitation of general-purpose data processing on ciphertext requires the provision of a predetermined, limited set of computational primitives, without imposing restrictions on the order and frequency of the primitive invocation. Consequently, this engenders considerable difficulty in controlling the computations conducted.

While a minimal set of primitives (e.g., addition and multiplication) may theoretically satisfy the constituents of arbitrary computable functions, as explicated in Section 2.4.1, the determination of practical computational capabilities for a given set of primitives remains elusive, due to the dearth of theoretical instruments capable of demarcating the boundaries of feasible computations that can be accomplished within an acceptable level of complexity. Thus, it is inconclusive which computational tasks can be deemed viable.

Moreover, even with the ability to reliably document and trace the historical record of primitive invocations, the low comprehensibility nature of these records poses a challenge to effective auditing. Imagine auditors have to check the compliance of current computational logic (like using certain attributes of underage individuals for joint analysis) based on circuits made up of addition and multiplication gates, or sequences of basic instructions. Such a method is generally considered unacceptable by auditing professionals.

Overall, SDP-based approaches are inadequate in addressing the exigencies of corporate compliance, risk management, and regulatory stipulations, in terms of their poor support for post-auditing.

3 TGCB APPROACH

In this paper, we propose an approach to alleviate the burden on the data owner's trust in the data processing procedure, termed "trust shifting". Our approach fundamentally changes the dynamics in delegated computation; previously, the data owner had to trust that data processing programs sufficiently maintained data confidentiality. However, we shift the trust to the interpreter, which is accountable for executing the data processing programs. Our proposal enables reliable data confidentiality for any acceptable programs, once trust in the interpreter for secure computation is confirmed. This holds true even when potentially malicious programmers have written the code.

The advantages of our approach are manifold. Theoretically, by relocating trust to the programming language interpreter, data processing confidentiality becomes independent of the computational logic, satisfying a general-purpose SDDP. In practice, trusted verification of the interpreter relies on a finite code base, providing cost-effectiveness compared to the infinite computational logic inherent in data processing. Once in place, this trust transfer mechanism endures over time. The subsequent subsections outline our proposed **TGCB**, a TEE-based computational backend that offers a truly general-purpose SDDP and is practical to handle real-world data processing applications.

3.1 Rationale and Tradeoffs

The establishment of "trust shifting" implies the existence of a programming language - EPL, the interpretation of which is acknowledged by the data owner, ensuring that the confidentiality of the data is not compromised when performing arbitrary scripts. So-called trust encompasses three levels of trust: design, implementation, and execution, which are mutually constraining factors.

To ensure data confidentiality when executing EPL programs, **TGCB** implements its core computation kernel with an EPL interpreter as the TEE's trusted code. Such a kernel is responsible for safeguarding data keys within the enclave and ensuring that sensitive data is not revealed outside the TEE. When computing sensitive data, **TGCB** decrypts the input ciphertext using its stored data keys, requesting the data owner if the key is not available. Additionally, **TGCB** also encrypts all data, no matter intermediate or final results, prior to dumping them from the EPC.

As mentioned earlier, TEE only provides integrity guarantees, and the safety of the trusted code itself needs to be ensured by the user. In other words, as with any other application designed for TEE, the data owner should perform a trustworthiness review of the implementation of **TGCB**. It must be acknowledged that although investing in reviewing the implementation of an interpreter is a one-time effort, it has economic advantages compared to repeatedly reviewing each computational task. However, the implementation of a language interpreter is typically more complex than a single simple computational task, which poses practical challenges in transferring trust to the compiler. Specifically, the language design and implementation of the EPL require a trade-off among the factors of ***minimality***, ***sufficiency***, and ***stability***.

- ***Minimality***: In the context of TEE applications, a fundamental consideration is to minimize the *Trusted Computing Base* (TCB) size, where the complexity of trusted code plays a crucial role. Therefore, it is important to simplify the implementation of **TGCB** as much as possible. This approach has two benefits: reducing the attack surface and minimizing the code review workload for the data owner. To achieve this goal, we carefully select the supported functionalities integrated into the EPL language design, which is explained further in Section 4.1. Additionally, in Section 5, we provide a detailed explanation of the implementation details of an interpreter that operates within an enclave and adheres to the EPL language design. This interpreter is designed to be concise, highly readable, and self-contained, implemented using C++ without relying on third-party libraries.
- ***Sufficiency***: The core idea of **TGCB** is to achieve a higher level - ***language*** - of secure delegated execution compared to primitives, by leveraging the security guarantees provided by TEE, as an answer to address the challenges mentioned in Section 2.4. In summary, **TGCB** serves as a computational backend that accepts "descriptive documents" of the data processing tasks expressed in the EPL, and performs the tasks while maintaining data confidentiality. EPL should be ***Turing-complete***, allowing the execution of arbitrary computable functions. Meanwhile, EPL scripts, which function as descriptive documents, should be highly readable to facilitate function-level audits instead of relying on lower-level forms, such as primitive sequences. In Section 4.1, we discuss the design of the EPL language, ensuring Turing completeness and other necessary functionalities for readability. Furthermore, in Section 6, we present a case study that applies **TGCB** to Spark, demonstrating direct operations on encrypted data in existing Spark applications, thus confirming its feasibility in enhancing the data security of real-world data processing.
- ***Stability***: The driving force behind the pursuit of trust shifting is based on the economic considerations arising from the prior review of TEE applications in delegated computation. This review ensures the implementation of trusted code for **TGCB**, which, once completed, can persistently serve the general computing of SDDP. This implies that **TGCB**, as an established trust foundation, needs to remain stable and cannot be frequently updated or patched. Otherwise, the entire code review process would have to start from scratch, undermining its original intention. To achieve this, the design of **TGCB** mentioned earlier strives to balance simplicity and functionality, ensuring that its implementation remains stable after a thorough examination. In Section 4.3, we also introduce extensions as an optional feature of **TGCB**. Their existence is driven by deployment considerations, allowing additional native functions to be flexibly plugged in when needed. However, this action

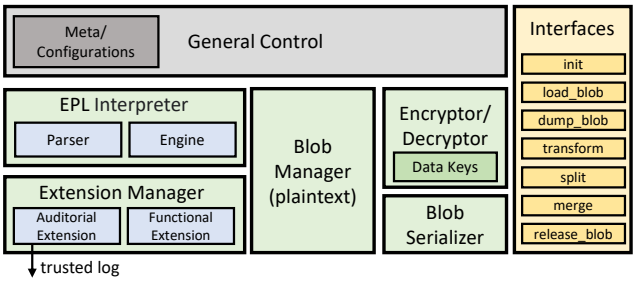

Fig. 3. **TGCB** architecture overview.

does not compromise the integrity of the **TGCB** implementation that has undergone a review, facilitating flexible deployment with a stable trusted code base.

3.2 Architecture Sketch

We first elaborate on the system architecture of **TGCB**, as shown in Fig. 3. **TGCB** loads ciphertext into the EPC for decryption and then manipulates the data by executing EPL scripts that support general-purpose computation. **TGCB** is designed as the trusted code for execution in TEE, and it exposes several **ECall**s as external interfaces. These interfaces support the main functionalities, including initialization, loading ciphertext into the EPC, receiving and executing EPL scripts, and dumping and releasing ciphertext out of the EPC. The Blob Manager is responsible for managing the sensitive data loaded into the EPC. When data is passed in as an encrypted byte array, it will be decrypted and deserialized into objects by the decryptor and blob serializer, respectively, and subsequently utilized as input for EPL script execution.

The EPL interpreter is a core component of **TGCB**, which parses and executes the EPL scripts in TEE to carry out arbitrary computable functions on sensitive data. In this process, the computational logic is described in EPL scripts, and the inputs and outputs are in the format of blobs. Besides, there are three cases: 1-1, 1-N, and N-1, corresponding to transforming, splitting, and merging interfaces, respectively. We shall present the implementation details of the EPL interpreter in Section 5.

The Extension Manager is tasked with administering **TGCB** extensions, which primarily comprise two categories: functional extensions and auditorial extensions. The former enhances the functions that may be invoked by EPL scripts, while the latter tailors the preconditions of EPL execution to accommodate practical necessities (e.g., communicating with external trusted logging services).

With the synergy of these components, once **TGCB**'s implementation is verified, loaded into the EPC as an enclave, and initialized properly, it provides a runtime at the granularity of script execution that manipulates the ciphertext data in a configurable and controlled manner.

3.3 Threat Model

Following the concept of SDDP, **TGCB** ensures ***data confidentiality*** when the data owner delegates their data to untrusted unions for data processing. Due to the code integrity guarantee provided by TEE, ***malicious adversaries*** cannot effectively interfere with the security mechanism of **TGCB**. We target a strong adversary, the malicious untrusted union with coordinator and executor, who can collude with each other. In particular, the coordinator can initiate arbitrary non-predefined or review-requiring data processing tasks, while the executor has privileged access to the computing resources: it can make full use of the OS capability to disrupt, manipulate, or hijack the data processing tasks that are running, and can also monitor/tamper with the content of hardware resources, e.g., memory, disk, and network. However, the adversary cannot access enclaves created

by TEE, i.e., data and execution inside an enclave are protected with respect to both confidentiality and integrity. In this adversarial scenario, there are no assumptions of goodwill toward untrusted unions, which is a more stringent condition compared to the typical honest-but-curious adversary who merely observes states passively.

We emphasize that **TGCB** does not depend on the coordinator to be trusted for data processing. This means that the programs used for data computation need not be concerned about being harmful. The untrusted unions, i.e., SDDP providers, have the freedom to decide what computation to perform without the need for the data owner's involvement. Moreover, in addition to safeguarding against SDDP providers exposing data through "malicious services," **TGCB** also encompasses the detection capability against "passive services" (e.g., denial-of-service or false service), enabling data owners to monitor actual computations performed on their encrypted data to ensure that data processing aligns with their expectations.

We exclude TEE side-channel attacks [30], since these vulnerabilities are mostly implementation-specific (e.g., SGX leaks memory access patterns). In the context of TEEs, side-channel attacks are usually "passive". Several related studies have proposed defense-enhancing techniques orthogonal to our approach, e.g., building the memory layer of the EPL interpreter on an oblivious memory model [72, 79, 92], in cases where the corresponding performance costs are worth. We also acknowledge "active" side-channel attacks, as EPL scripts written by untrusted programmers can potentially facilitate such attacks, encoding sensitive data through execution behaviors, such as time and output size. Apparently, aided by the post-hoc auditing capability of **TGCB**, it is easy to detect the execution behaviors of such EPL scripts with the intention of stealing data. However, **TGCB** still suffers from limitations in that it currently lacks any effective preventive defense mechanisms, bringing risks of data leakage. To alleviate this, a possible strategy is to add noise in **TGCB**, e.g., (i) introduce uncertain latency when executing EPL statements, (ii) mix dummy into ciphertext before it leaves TEEs, so that the signal-to-noise ratio of encoding the plaintext is lowered. Exploring the possibility of establishing additional security assumptions with theoretical guarantees should also be considered as part of future research efforts.

4 SYSTEM DESIGN

4.1 Encryption Programming Language (EPL)

To support general-purpose computation as a secure computation backend on a language level, the naïve solution is to integrate a modern language runtime directly into TEE. Nevertheless, this not only results in an overly large trusted code, but also has three major drawbacks. First, it is highly difficult for data owners to initiate an effective code review to ensure that there are no malicious implants or underlying vulnerabilities. Second, the overly rich functionalities pose a serious challenge in determining the boundaries of the supported behaviors and possible operations, e.g., disclosing sensitive data directly by undefined behaviors. Third, frequent updates to the runtimes of modern programming languages are necessary to patch vulnerabilities that arise from their inherent complexity. This renders the trusted code that integrates them inherently unstable, preventing it from being exposed to known attack patterns. In order to overcome such limitations and support arbitrary computable functions, **TGCB** proposes the use of a scripting language, EPL, to capture the computation logic to be executed on decrypted data, the runtime of which is ***minimal***, ***sufficient***, and ***stable***.

EPL, as a descriptive document for cryptographic data processing tasks, serves the fundamental purpose of describing a computable function that is expected to transform plaintexts. Its focus is on the operations performed on the data and the control flow of logic, rather than providing other functionalities, such as scheduling hardware resources for process management, memory

allocation, and I/O operations. Consequently, the computational logic of any EPL script can be regarded as a function with a single input and a single output channel, ensuring that function execution does not involve intermediate side effects that could lead to data leakage. To support the representation of arbitrary computable functions in EPL, the language needs to be Turing-complete. This means that the language should, at a minimum, support basic data types and their algebraic and logical operations, allowing for control flow capabilities equivalent to a Turing machine.

The simplest Turing complete control flow paradigm, which FORTRAN relies on, is `if-goto`. However, EPL has chosen not to incorporate `goto` into the language. The reasons are two-fold: firstly, `goto` is not a commonly used method for expressing computational logic from today's perspective, particularly in higher-level computation where it could easily introduce subtle logical errors that are difficult to detect; secondly, `goto` does not facilitate the creation of a more easily auditable descriptive document.

In the absence of conditional jumps in the control flow, there are two paradigms that are also Turing complete: (a) λ-calculus [14], with LISP being a typical implementation that achieves this through supporting function definitions and conditionally recursive calls [47]; (b) structured program theorem (a.k.a. Boehm-Jacopini theorem [16]), encompassing conditional infinite iterations (`while-if`), which is widely used in most imperative programming languages today. EPL chooses to support both paradigms because they are widely adopted in existing data processing tasks. Although it is possible to rewrite either paradigm equivalently, such a transformation might be nondirect and less readable.

Meanwhile, we choose to introduce flow statements, that is, `continue`, `break`, `return`, as compensation for not introducing `goto`. Otherwise, when expressing certain existing functions depending on these flow controls in EPL, we would have to use nested conditional blocks, which significantly compromises readability. For a similar reason, we also choose to support structural variables to avoid cumbersome token flattening. However, we choose not to introduce other more complex modern language features such as exceptions, reflections, object-oriented programming, etc. We believe that the absence of these does not hinder the effective expression and auditing of computational functions, as our guiding principle is minimality.

In summary, to ensure the ease of interpreter implementation as well as the usability of the programming language, EPL supports the following five syntaxes:

(1) Conditional branching (`if-else`)
(2) Function defining (with early `return`)
(3) Recursive function invoking
(4) Loop (`for`, `while`, `break`, `continue`)
(5) Composite structural variable

These features enable EPL to adhere to the conventions of imperative programming, facilitating the migration of extant computational logic. Furthermore, EPL satisfies the need for complex and dynamic computational flow, which cannot be expressed in a data-independent manner.

Further, given that such a script interpreter will be executed in TEE, which generally provides the C/C++ SDK, EPL is designed as a C++-embedded scripting language with its interpreter execution environment and the native code interoperating in both directions. Concretely speaking, the interpreter receives a script string by providing C++ interfaces, executes it, and then returns the results. Meanwhile, the script interpreter also provides C++ interfaces to allow the native code to incorporate the references of variables and functions into the interpreter's execution environment, thereby allowing them to be utilized by the scripting language. In this way, the EPL serves as a glue language to invoke C++ functions, enabling intricate control flows that rely on conditional branching. The security of EPL is bounded because it only provides the ability to control the flow

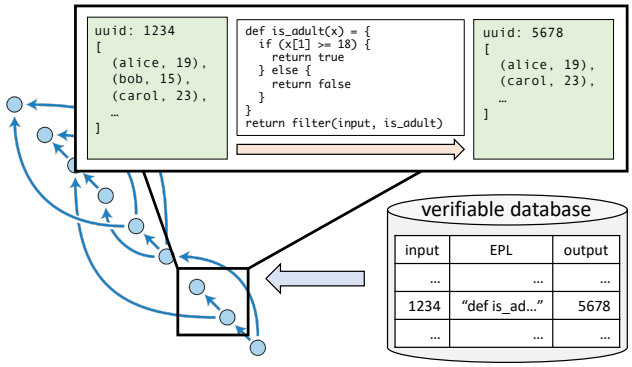

Fig. 4. **TGCB**'s DAG computational flow model.

of computation on visible plaintext, which addresses the drawbacks of existing primitive-based approaches that must perform computation in a data-independent manner.

4.2 Computational Flow Model

TGCB manages sensitive data as blobs that reside in plaintext format within the EPC, and executes EPL scripts with these blobs serving as inputs and outputs. Each data blob is uniquely identified with a *Universally Unique Identifier* (UUID) and may originate from decrypting an encrypted byte array loaded externally to the TEE or from the output of an EPL script execution.

In Fig. 4, we show an example of executing an EPL script for a transformation logic, i.e., a 1-1 computational task that takes one UUID as the input, and returns one blob's UUID as the output. In this example, given a blob (UUID: 1234[1]) with a list of pairs (name, age), our aim is to run an EPL script to select all adults (age>=18), so the script execution will generate a blob with another UUID assigned, e.g., 5678, recording the output result in the EPC. Then the execution behaviors of an EPL script correspond to a triplet that is composed of: the UUID of the input blob, the EPL script string, and the UUID of the output blob, respectively. We can log this triplet in a trusted way, i.e., let logs persist on verifiable databases, to ensure that every EPL execution is recorded. As an active research direction, the design of verifiable databases is orthogonal to **TGCB**, with typical solutions such as blockchain-based [29, 31] and TEE-based [104], in order to guarantee that all logs persist successfully and are tamper-proof. Now, if we treat the blobs as nodes and the EPL scripts as edges, the trusted logs consisting of triplets will constitute a *Directed Acyclic Graph* (DAG). One special case is that when the data owner encrypts the data into a blob, **TGCB** will assign a UUID to this blob and record this encryption behavior in trusted logs, as a node without in-degrees.

We note that the design of **TGCB**'s DAG computational flow is crucial. First, the complete history of any manipulation of the ciphertext is secured, which delivers full data provenance support. More precisely, the combination of trusted execution and trusted logging ensures that any data manipulation via EPL is authentically recorded. This further brings **TGCB** the capability of post-hoc auditing on the basis of non-repudiation evidence. In addition, it is straightforward to support reachability queries on DAGs. This, thus, renders all the dependency and consumption of a particular ciphertext definite in SDDP. Finally, as all EPL scripts are recorded in the trusted logs as edges, **TGCB**'s DAG computational flow model is also equipped with the capability of replay/recovery, which enables the verification of data integrity.

[1]The UUID format is simplified for illustration, the same below.

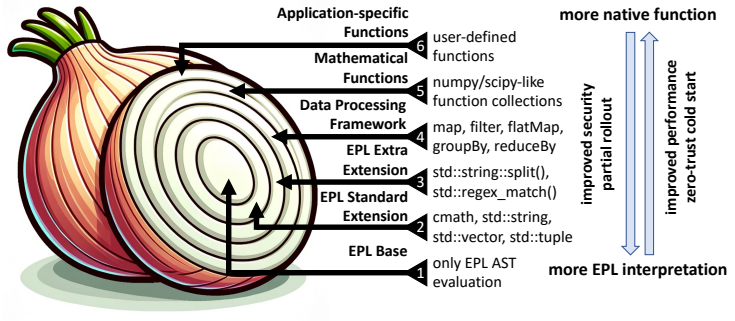

Fig. 5. Onion model of EPL extension.

4.3 Extension Model

In this subsection, we introduce an optional feature provided by **TGCB**, the extensions. Grounded on the design rationale of the EPL's functionalities described in Section 4.1, the interpreter itself does not build in any functions and supports merely five kinds of syntaxes. Theoretically, any computable functions can be written in EPL and then executed in TEE. In other words, any computational tasks can be accomplished by interpreting the execution of EPL-only scripts. As a result, the execution process is fully trusted. Nonetheless, from a practical perspective, the EPL interpreter can also import libraries for the EPL scripts on extra functions via the extension. Therefore, when the EPL interpreter reaches a function call statement, the corresponding function body can be either an *abstract syntax tree* (AST) parsed by an EPL function definition or a native function wrap. There is a clear trade-off between the two: while EPL scripts offer higher reliability without necessitating reviews on implementations, they are slower to execute than the native code for the same computation. Moreover, stringently requiring all code to be written in EPL scripts would be detrimental to the wide deployment/adoption of **TGCB**.

Considering all the factors above, we propose to abstract the functions that are required in general-purpose SDDP in a hierarchical manner, i.e., the onion model of the EPL extension for function calls in the EPL TEE runtime, as illustrated in Fig. 5. The EPL base refers to the EPL interpreter without any extensions, with which, as the core, the EPL interpreter can be extended layer by layer to enrich the native functions that can be called. The examples of typical functions in each layer are also given in the figure.

As an example, let us consider the computational task of filtering out all prime numbers from input data, with the corresponding EPL script shown in Fig. 6. In line 25, this task is completed by calling the filter with the input data and a Boolean expression. We first dive into the details of the function `is_prime` as defined in line 1, which determines if a number is prime. For the sake of this example, we adopt the implementation of na"ive enumeration, which involves the `sqrt` function (line 2) to compute the square root, in addition to the basic loop and conditional branching. On the premise that there are no extensions, we need to implement `sqrt` by defining the EPL function on line 20. Further, in lines 7-19, we implement a filter function that enables access to the filtered data after providing an input iterator and a Boolean expression to accomplish this task.

In this example, according to our onion model, `sqrt` belongs to the EPL Standard Extension, `filter` falls into the Data Processing Framework, whereas `is_prime` lies on the outer layer and can be regarded as a Mathematical Function or Application-specific Function. With the onion model, the basic trust is first established by the EPL base as a premise, and it serves as the core for further outward extensions. While maintaining the stability of **TGCB**'s trusted code, these extensions provide

```
def is_prime(x) = {
  for (var i = 2; i <= sqrt(x); ++i) {
    if (x % i == 0) return false
  }
  true
}
def filter(iter, func) = {
  def hd = nil
  def hdDefined = false
  def hasNext = {
    if hdDefined return true
    do {
      if (!iter.hasNext) return false
      hd = iter.next
    } while (!func(hd))
    hdDefined = true
  }
  def next = { if (hasNext) {hdDefined = false; hd }}
}
def sqrt(x) = {
  var y = x
  while (! y * y - x < 1) { y = (y + x / y) / 2 }
  y
}
filter(input, is_prime)
```

Fig. 6. EPL script example of "prime number filtering".

the capability to include additional native functions. It is crucial to highlight that executing these native functions inside TEEs can expand the potential attack surface. This necessitates a degree of confidence when integrating native code into the extension payloads. Therefore, incorporating new extensions hinges on either a comprehensive evaluation or the data owner's acknowledgment of the related risk.

Nonetheless, this design allows the **TGCB** framework to detach itself from a fixed reliance on the trustworthiness of extensions, the adoption of which can earn additional bonuses on practical deployment considerations, i.e., the use of extensions allows for dynamic adjustments in collaboration within SDDP. To be specific, data owners and SDDP providers can then determine the appropriate level of extension in the onion model based on specific application scenarios, cooperation modes, and other relevant factors. By selectively configuring the extension layers, they can optimize the system set-ups for practical use. Meanwhile, the onion model also provides space for flexible adjustments in dynamic partnerships, e.g., a partial rollout of attaining the target extension level. For instance, SDDP can be initiated from the outermost layer of the onion model. For existing computational tasks, with the primary goal of boosting data security, we can start by wrapping most functions with existing static library implementations and then gradually express part of the corresponding logic in EPL. In this way, we reduce the unnecessary use of native code to achieve improved security. In another direction, SDDP can also be initiated without any consensus of trusted code except **TGCB**, and serve computational tasks solely through EPL scripts without native functions. In this process, users would begin by developing and reviewing EPL extensions that contain native function implementations, gradually expanding the trusted function calls into the runtime for execution in a native manner.

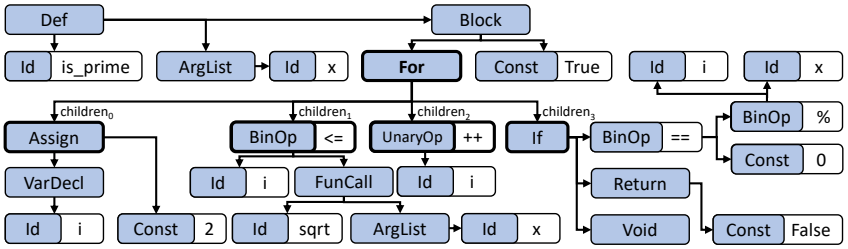

Fig. 7. EPL AST of the `is_prime` function.

5 EPL INTERPRETER IMPLEMENTATION

This section provides implementation details of the EPL interpreter, showcasing how it achieves simplicity, intuitiveness, and ease of verification and trust for data owners, while adhering to **TGCB**'s design philosophy regarding necessary language features and security boundaries. Our EPL interpreter is written entirely in C++ and utilizes only the *Standard Template Library* (STL), ensuring complete control of implementation and minimizing the risk of hidden security vulnerabilities like underwater icebergs due to the dependencies of third-party libraries. The interpreter consists of three main parts: the AST node for representing the computational logic, the boxed value for holding the data, and the execution context.

5.1 AST Node

`AstNode` is an essential data structure of the EPL interpreter, which denotes a tree node of EPL AST. As a base class, `AstNode` includes the following primary members: (i) an enumerated type of the node, (ii) a vector of references to its children nodes, (iii) a string containing auxiliary information, and (iv) a virtual function returning a boxed value, which serves to evaluate the interpretation execution results of the AST corresponding to a tree node. Hence, an AST node can also denote the subtree rooted in itself, i.e., a corresponding series of computational programming logic.

When the EPL script is input as a string, the EPL parser will process it, construct the script's corresponding AST which is composed of several AST nodes, and finally return the root node. For example, for the `is_prime` function in Fig. 6, the parser will build a corresponding AST, as shown in Fig. 7. The types of `AstNode` are shown in colored boxes, and some are accompanied by white boxes indicating extra information in a string. The EPL interpreter's execution of scripts is determined by the traversal of the AST. Specifically, when evaluating a subtree with an `AstNode` as the root, the interpreter implements the `Eval` virtual interface, depending on the node's derived class type, and traverses its child nodes in a specific manner. For the example `ForAstNode` (i.e., `AstNode` for "`For`" in Fig. 7), we display the partial code implementation of the base class `AstNode` and its derived class `ForAstNode` in Fig. 8. The syntax of the `for` loop in EPL is the same as in C++; that is, `ForAstNode` has four child nodes: three statements ($\text{children}_{0,1,2}$) and one loop body (children_3), which are in bold font in Fig. 7. We declare a local variable `i` in children_0 (line 11) and evaluate children_1 before entering the loop (line 12). If the result of casting to `bool` from the evaluation of $\texttt{children}_1$ is `true`, then the loop body, which is $\texttt{children}_3$, is evaluated (see line 15). After every execution of the loop body, $\texttt{children}_2$ is evaluated (see line 13) to manipulate the variable `i`, influencing the result of $\texttt{children}_1$'s `Eval`.

In addition, the EPL interpreter turns to the exception mechanism in C++ for event passing between nodes when traversing AST. The children_3, as the loop body, is expected to throw two types of exceptions, `Continue` and `Break`. During the traversal of the children_3 subtree, these two exceptions will be thrown upon encountering `ContinueAstNode` and `BreakAstNode`, and caught

```
1  class AstNode {
2    const AstNodeTypes type_;
3    const std::string extra_;
4    std::vector<std::reference_wrapper<AstNode> > children_;
5    virtual BoxedValue Eval(ExecutionContext &ec) const = 0;
6  }
7  class ForAstNode final : AstNode {
8    BoxedValue Eval(ExecutionContext &ec) const override {
9      ec.PushStack();
10     try {
11       for (children_[0].Eval(ec);
12            CastBoolCondition(children_[1].Eval(ec));
13            children_[2].Eval(ec)) {
14         try {
15           children_[3].Eval(ec);
16         } catch (exceptions::Continue &) {}
17       }
18     } catch (exceptions::Break &) {}
19     ec.PopStack();
20     return BoxedValue.Void();
21   }
22 }
```

Fig. 8. Code snippet of `AstNode`.

by `ForAstNode` inside the loop (line 16) and outside the loop (line 18), respectively. After the exceptions are caught, no more operations are required, and the expected control flow is thereby achieved. There are also other types of `AstNode` involved in Fig. 7, such as another crucial branching syntax `IfAstNode`, the implementation of which greatly resembles that of `ForAstNode`. Due to space limitations, we will not delve into them further.

5.2 Boxed Value

`BoxedValue` serves as a versatile data container within the EPL interpreter. It is, by nature, a wrapping of C++ variables (via a pointer `void*` and `std::type_info` that can be either a C++ built-in type or any C++ class). `BoxedValue` serves as a uniform data format when executing EPL scripts, i.e., evaluating AST.

Among `BoxedValue`'s supported types, there are two wrapped classes that we need to highlight. One is the wrapping of callable, i.e., `BoxedValue` holds a `std::function` object pointer, which involves two cases: (i) a native function as described in Section 4.3; (ii) a function defined by EPL, the execution of which is then in nature `Eval` of `DefAstNode` parsed from the function definition.

The other is the wrapping of `epl::DynamicObject`, which maintains a `std::map<std::string, epl::BoxedValue>`, to represent the mapping between the member names and `BoxedValue` in a compound type. `DynamicObject` handles all compound types in the EPL interpreter; in other words, the dot (`.`) operator is used for direct member selection via the object name. As for children_0 of `DotAccessAstNode`, the `BoxedValue` of the result returned by its `Eval` should wrap `epl::DynamicObject`. According to the `Eval` result of children_1, we then direct the child object for access in the mentioned map, regardless of whether it is just a variable or a callable. In the EPL interpreter, all objects of supported dot access operators are processed in this manner, e.g., (i) a compound structural variable defined in EPL discussed in Section 4.1, and (ii) a C++ class object

```
class EncMapPartitionsRDD[U, T] (
    prev: EncRDD[T], f: EncIterator[T] => EncIterator[U])
    extends MapPartitionsRDD[U, T] with EncRDD[U] {
  def compute(): EncIterator[U] = prev.compute().map(f) }
class EncIterator[T] (
    var payload: Either[TEEEntry[T], EncTransform[T, _]])
    extends Iterator[T] {
  def map[U](f: T => U): EncIterator[U] = {
    val transf = (iter: Iterator[T]) => {
      new AbstractIterator[U] {
        def hasNext: Boolean = iter.hasNext
        def next(): U = f(iter.next()) }}
    new EncIterator[U](
      new EncIteratorTransform[U, T](this, transf)) }}
class TEEEntry[T] (var blob: UUID) extends Serializable {}
class EncIteratorTransform[U, T] (
    val prev: EncIterator[T],
    val func: Iterator[T] => Iterator[U]) {}
```

Fig. 9. Code snippet of Spark backend switching.

wrapped by `BoxedValue` as variables in EPL, the members of which can be accessed via the dot operator in EPL.

5.3 Execution Context

The EPL interpreter requires an `ExecutionContext` to be maintained during runtime. There are three types of context information for the EPL interpreter, namely ***Functions***, ***Stacks***, and ***GlobalVariables***, which are wrapped in `epl::ExecutionContext`. When executing `Eval` of `AstNode`, as shown in line 8 of Fig. 8, this context information will be passed as an argument.

Functions maintain the mapping between names and definitions. When evaluating `FunCallAstNode`, ***Functions*** further link the `DefAstNode` given a function name to execute the function call.

Stacks maintain local variables at the level of scope through `stack< std::map<std::string, epl::BoxedValue> >`. Each element in the stack denotes a scope. When declaring a local variable, the pair of its name and `BoxedValue` will be inserted into the map on the stack top (the current scope). When searching for a variable given the id token, we will start from the top of the stack, that is, we will traverse the scope levels inside out to check if the corresponding variable name and `BoxedValue` exist in these maps.

GlobalVariables share a high similarity with ***Stacks***, except that there is only one scope for maintaining global variables (e.g., `global pi = 3.14`). Explicit declarations of global variables (`GlobalDeclAstNode`) will directly add the variable value into global variables, so that its lifetime is not impacted by the scope of the declaration statement. Besides, EPL does not distinguish between global and local when searching for a variable. Instead, we search in ***Stacks*** in sequence first; if the variable is not found, we further search in ***GlobalVariables***.

6 CASE STUDY: SPARK PLUGIN

In this section, we present the **TGCB**-enabled Spark plugin as a case study. Such a plugin enables untrusted coordinators to leverage existing Spark applications to support SDDP seamlessly, without requiring any modifications. This is particularly valuable considering the widespread usage and well-established ecosystem of Spark, highlighting the engineering significance of **TGCB** and the adequate functionality provided by the EPL. Due to space constraints, we refer interested readers

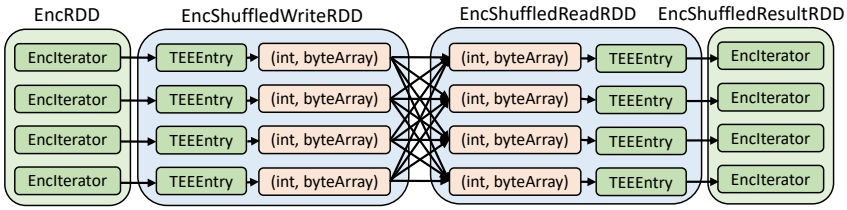

Fig. 10. Wide dependency topology of `EncRDD`.

to [99] for a detailed technical overview of Spark. We focus on the implementation details of three major components in the Spark plugin for **TGCB**.

6.1 Computational Backend Switching

We replace Spark's computational backend with **TGCB**, modifying its data manipulation functions to accommodate ciphertext transformations, all while preserving Spark's distributed features. Spark handles data as *Resilient Distributed Datasets* (RDDs), and processing these datasets necessitates dependence on multiple parent RDDs. These dependencies can be categorized as ***narrow*** and ***wide***.

Narrow Dependency. We use the ***map*** operator as an example shown in Fig. 9. Spark uses `Iterator` to access materialized results in the RDD abstraction. We have incorporated an `EncRDD` trait into every RDD that represents the ciphertext. As a result, we have `EncMapPartitionsRDD` that abstracts the map transformation results on the parent RDD. `EncIterator` is used to represent the ciphertext. It has two possible payload forms: `TEEEntry` or `EncIteratorTransform`. `TEEEntry` signifies a blob managed by **TGCB** with a UUID identifier. `EncIteratorTransform` indicates data yet to be converted into a blob and should be abstracted using the parent `EncIterator` and the transformation function.

Wide Dependency. This involves data shuffling or redistribution, managed by `partitioner[T->Int]` in Spark, which assigns a partition ID to each record. Handling `EncRDD`'s wide dependency requires the partitioner to run in **TGCB** via EPL, as it needs access to each record's plaintext. The transformation of `EncRDD`'s wide dependency results in three RDDs in Spark's DAG is illustrated in Fig. 10. `EncShuffledWriteRDD` materializes the parent `EncRDD`'s `EncIterator` into `TEEEntry`. Through **TGCB**'s split interface, the partitioner's logic inputs are given, dividing the blob into multiple blobs. We serialize and encrypt each blob into a byte array, shifting it from EPC to JVM memory, forming an `RDD[(Int, ByteArray)]`. Spark's original functionalities shuffle `EncShuffledWriteRDD` according to `Int`, ending with `EncShuffledReadRDD` after shuffling. The `ByteArray` is transferred to the same partition, creating multiple blobs, and merged into a `TEEEntry` through the merge interface of **TGCB**. `EncShuffledResultRDD` recovers `TEEEntry` to `EncIterator`, producing a new `EncRDD`, marking the completion of the `EncRDD` shuffling.

6.2 EPL's Spark Transformation Translator

When handling narrow and wide dependencies, due to the requirement of accessing the plaintext, the execution of `f` and `partitioner` should be conducted in **TGCB** via EPL. As Spark is a sophisticated large-scale data processing platform with a mature application ecosystem, we develop an EPL Spark transformation translator to leverage this advantage. This translator enables the execution of Spark applications on ciphertext based on EncRDD without explicitly writing computational logic in EPL, allowing for the continued use of existing application codes (in Scala). Specifically, for Scala-written functions (i.e., lambda expressions), we obtain the JVM bytecode (*.class* file) of the function body through `ClassLoader`. Subsequently, after disassembling and decompiling

the bytecode, we generate the corresponding Java AST and further translate it into EPL scripts. As a result, the translator converts Scala functions into EPL scripts for execution within **TGCB**. This approach facilitates the direct utilization of existing Spark applications written in Scala on ciphertext in **TGCB** with "zero-modification"[2].

6.3 Unified Storage Management

Recall that the sensitive data maintained by **TGCB** in the EPC is identified and managed by Spark as `TEEEntry`. In Spark's original storage management, data storage assumes various forms depending on RDD's storage levels, e.g., Java objects and serialized byte arrays. Through the implementation of relevant interfaces for `EncRDD` and `TEEEntry`, the Spark plugin of **TGCB** adapts ciphertext to the Spark platform, achieving unified storage management. To be specific, when Spark memory management deserializes an RDD, `EncRDD` loads the byte array from JVM to EPC through the load_blob interface of **TGCB** for decryption, and finally returns the corresponding blob UUID forming a `TEEEntry`. On the contrary, when the serialization interface is invoked, `EncRDD` will access the UUID corresponding to `TEEEntry`, based on which **TGCB** will serialize and encrypt sensitive data in a byte array and transmit it from EPC to JVM. In addition, the data in the memory is stored in different memory modes (e.g., `ON_HEAP` and `OFF_HEAP`). On account of this, we add another memory mode `ON_TEE` in order to manage `TEEEntry` stored in the EPC in Spark.

7 EVALUATION

We conduct extensive experiments[3] to demonstrate that **TGCB** provides a practical supplement to the universality of SDDP. To our knowledge, there are no perfectly suitable counterparts of **TGCB**. Hence, we first examine the EPL execution performance of **TGCB** in a view as a language interpreter to clearly demonstrate the performance cost it incurs for additional security properties. Then, we compare **TGCB** with the most relevant studies from several research directions related to confidential computation.

7.1 EPL Interpretation Performance

Introducing security guarantees in computations comes at a high cost, including storage and communication scale, deployment complexity, and especially increased execution time, which generally results in a significant performance overhead of several orders of magnitude. To provide readers with a clear understanding of the performance costs associated with using **TGCB** in general-purpose data processing for enhanced confidentiality, we compare EPL with Python, an established interpreter that works on plaintext in a traditional insecure environment, and the experimental results are shown in Fig. 11. The computable functions selected are (a) the Miller-Rabin test [51], (b) Pi estimation by the Monte Carlo method [50], and (c) K-means clustering [46]. We select three typical benchmarking tasks for general-purpose computation[4] that are challenging for traditional confidential computational approaches due to their reliance on (pseudo)randomness and data-dependent conditional branches. In the meanwhile, for each evaluated application, we select a key component[5] to load into the EPL interpreter runtime via an extension, so that the relevant

[2]The translator is only applicable to the lambda expressions of which all computational logics are expressed in JVM bytecode, and its output includes a complete implementation of all computations in EPL, without relying on any **TGCB** extensions.
[3]All experiments are conducted on a machine with Linux 4.19.91, equipped with Intel(R) Xeon(R) Platinum 8369B CPU @ 2.70GHz and 32GB memory.
[4]In particular, the detailed workloads are: (a) 3 times of Miller-Rabin primality test on each positive integer that is smaller than 5×10^4; (b) uniform generation of 10^5 points in the unit square to record the frequency of falling in the unit circle area; and (c) 10 K-means iterations on the D31 [84] dataset.
[5]The embedded native functions are: (a) $a^b \bmod c$; (b) generate a random point in the unit square; and (c) find the nearest one among a set of points.

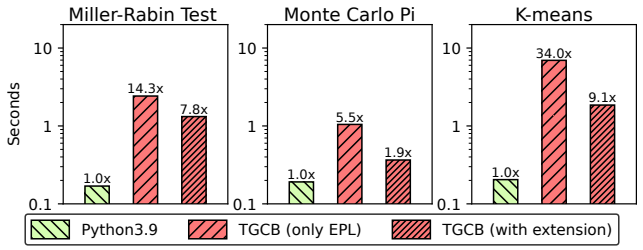

Fig. 11. Performance comparison with Python.

Table 1. Performance comparison between GoogleFHE and **TGCB**.

function / approach	TFHE	OpenFHE	**TGCB**
fibonacci number	60.9 secs	48.4 secs	< 1 ms
integer square root	71.1 secs	60.5 secs	< 1 ms
string capitalization	151.5 secs	118.8 secs	< 1 ms

functionalities are performed as native code. As observed from the experimental results, compared with Python, EPL is 5x-34x slower across applications when executed by **TGCB**'s interpreter in TEE. Further, loading common operators as native functions through extensions can speed up the execution of **TGCB** by about 2x-3x and make EPL scripts easier to program simultaneously. There exists an apparent performance gap between Python and **TGCB**, as the price for data confidentiality, which is due to (1) **TGCB** running in TEE, introducing inherent performance overhead, and (2) the performance of the EPL interpreter being less superior to Python. Besides the fact that Python is more mature and well-engineered, which integrates a number of optimization techniques, the compromise made by the EPL interpreter, facilitating a reliability verification of the trusted code, is also part of the reason. For instance, (i) compared with Python, which generates bytecode (`.pyc`) in advance for execution by the PVM, EPL employs AST traversal, i.e., the execution of each AST node relies on recursive calls to C++'s virtual interface; (ii) ID searching in EPL relies entirely on `std::map<std::string>` to complete the search for the corresponding boxed value, which introduces considerable overhead. Thus, such designs in EPL relatively sacrifice performance to a certain extent in order to make the interpreter implementation more concise, straightforward, and easy to review.

7.2 FHE-based General Computation

In terms of executing arbitrary computable functions on ciphertext, FHE-based general-purpose transpilers are the most suitable competitors, among which GoogleFHE [38] is a recent and advanced solution. GoogleFHE converts computable functions written in C++ into circuits composed of FHE gates, enabling the corresponding computation on ciphertext encrypted by specific FHE schemes. The comparison of execution performance between GoogleFHE and **TGCB** is shown in Table 1. The computable functions involved are three built-in benchmarking samples[6] of GoogleFHE, and **TGCB** implements them with exactly the same logics.

[6]In particular, these three benchmarking computable functions are: (a) calculating the 10th Fibonacci number; (b) computing the square root of 15875 (rounded down to an integer); (c) converting a fixed-length char array ("do or do not; there is no try") into uppercase, respectively.

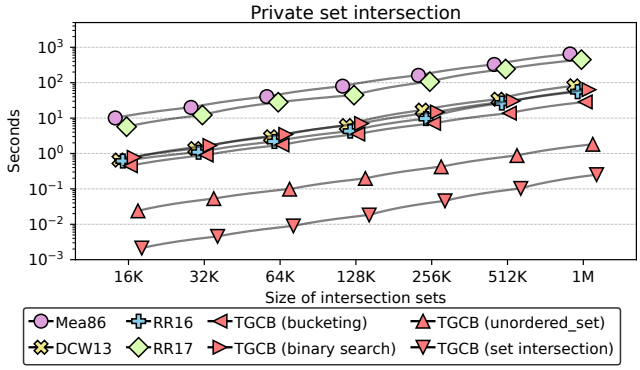

Fig. 12. Performance comparison among PSI solutions.

Specifically, we test both FHE execution backends, TFHE [22] and OpenFHE [6], supported by `GoogleFHE`. It is clear that the runtime comparison is not particularly meaningful, as it is not practical to execute arbitrary functions by constructing FHE circuits based on the latest available FHE schemes, even in the case of tiny data sizes. Conversely, our proposed **TGCB** presents a substantially more efficient alternative solution. Besides, there exists an explicit restriction on the C++ functions that `GoogleFHE` can translate, i.e., the computation represented by the FHE circuit must be data-independent. Since the gates cannot access true values as explained in Section 2.4.1, many syntaxes of traditional imperative programming languages are not possible, such as variable-length loops and arrays, recursion, and early return of functions. These limitations make it inconvenient for `GoogleFHE` to write real-world functions. On the contrary, EPL does not have such restrictions and hence is more practically useful.

7.3 Cryptography SCO Component

Given the significant performance limitations of extant tools based on perfect FHE, SCO solutions have been widely explored and applied as a compromise solution in privacy-preserving computing. Specifically, a particular SCO solution is typically composed of multiple key components and is operated by a trusted coordinator in a specific organizational manner, i.e., protocols. Due to the complexity and diversity of various protocols, as well as space limitations, we select one of the most representative components, *private set intersection* (PSI), for experimental comparison. PSI generally involves two parties, each holding a set, with the goal of obtaining the intersection of the two sets through joint computation without exposing non-intersecting elements.

We select several representative studies from this research line as baselines. `Mea86` [49] stands as one of the pioneering PSI protocols, which utilizes public-key cryptography and the multiplicative homomorphic attribute of the Diffie-Hellman key exchange. As the field evolves, more promising strategies based on *Oblivious Transfer* (OT) are conceived. In particular, `DCW13` [28] realizes an OT protocol by integrating Bloom filters, employing a semi-honest adversary. `RR16` [67] then improves and enhances this concept by extending the threat model to a malicious adversary with concessions. Furthermore, `RR17` [68] integrates hashing techniques and implements the standard malicious secure model. Performance comparison among these PSI solutions is shown in Fig. 12. In this comparison, we implement the same functionality using **TGCB** in four different ways: (a) hash table (bucketing and chaining); (b) binary search (via defining a function in EPL) on the sorted array; (c) extension with `std::unordered_set`; and (d) extension with a native function to calculate shared elements in two given arrays. We note that (a) and (b) are based on pure EPL, and (c) and (d)

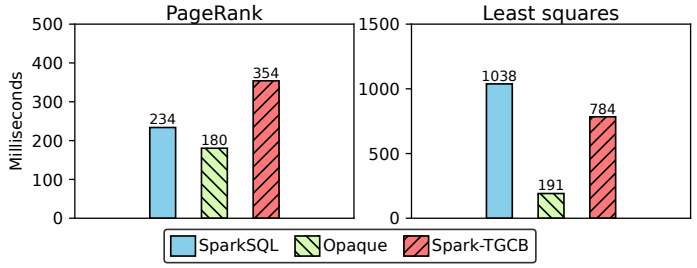

Fig. 13. Performance comparison on Spark tasks.

use extensions embedded with native code. To demonstrate the scalability of multiple evaluated approaches, we conduct experiments using different data sizes. As shown, baseline PSI solutions achieve relatively acceptable performance by application-specific optimization in a medium-sized workload. For **TGCB**-based methods, with EPL-only interpretation, the performance is comparable to that of the best-performing baseline, whether based on vanilla bucketing or binary search. Moreover, the implementations are quite straightforward compared to the sophisticated protocol design of application-specific PSI approaches. The performance of **TGCB** is greatly improved when completing PSI tasks with the standard container `std::unordered_set` loaded via extension. Further, if a close-to-native speed is desired, this can be achieved by loading a specific native function with trusted implementation. Meanwhile, compared to **TGCB** which targets a strong adversary as described in Section 3.3, PSI baselines introduce assumptions that weaken security, e.g., participants being honest-but-curious or certain leakage of data distributions.

7.4 SDP-based Secure Computation on Spark

In this experiment, we contrast **TGCB** with Opaque [102], an SDP-based solution on Spark. Although both rely on TEE, **TGCB** accommodates arbitrary functions, whereas Opaque is limited to SQL queries. We investigate two widely used database workloads [61], namely PageRank and Least Squares, which are also benchmark cases established in the Opaque. We conduct a comparative analysis of the performance of SparkSQL [3] (insecure), Opaque, and **Spark-TGCB**. The results are shown in Fig. 13. For **Spark-TGCB**, as described in Section 6, the normal applications written in Scala are submitted to Spark directly. After that, our EPL translator transpiles the specified tasks into EPL scripts, which are executed by **TGCB** as the computational backend. We observe that Opaque, being TEE-based, is similar to or even faster than SparkSQL[7], indicating that TEE-based SDDP schemes achieve superior engineering practicability over cryptography solutions. By implementing pre-defined SDPs in the trusted code, computational tasks that can be represented in a data-independent execution plan, such as PageRank and Least squares[8], can securely and efficiently process the ciphertext. In cases where such conditions are not ideally met, and there is a heightened requirement for universality that existing solutions struggle to fulfill, **TGCB** can serve as a timely alternative. Furthermore, since **TGCB** operates at a higher level of granularity, specifically focusing on functions rather than primitives, it can provide improved assistance for conducting post-hoc audits of the computations carried out.

[7]The main reason is that SparkSQL is executed by JVM, whereas Opaque is executed in TEE with C++ native code. The overhead introduced by JVM compared to native code is higher than the TEE cost in certain cases.
[8]However, it is not straightforward to do so in Opaque, since tasks need to be represented through SQL queries. In the case of PageRank, it projects data into a flattened table (src, dst, rank, weight) by `join` and `select` in advance and then completes an iteration through `groupby` and `aggregate`.

8 RELATED WORK

Privacy-preserving Computation. The concept of HE is introduced in 1978 [69] as an intuitive method for SDDP. Early HE research allows only a single operation type on ciphertext, such as addition [58], multiplication [70], and XOR [37]. The first FHE algorithm emerges in 2009 [32], theoretically supporting arbitrary functions. While FHE opened new possibilities, its performance overhead led to various optimization efforts [33, 59, 91]. However, achieving practical efficiency has been a persistent challenge. To mitigate this, researchers start to make compromises for practical performance: (i) application-specific solutions for different fields, e.g., Linear Equation [19], Matrix Multiplication [13], Matrix Factorization [103], Regression [55], and Aggregation [66]; (ii) semi-honest adversary assumptions [36], wherein participants will strictly follow computation protocols [20, 44, 95], preventing data leakage from observing intermediate results; (iii) secure computation protocols [88, 93] that permit certain knowledge disclosure.

TEE-based Security Enhancement. In order to render the security enhancements brought by TEE more pervasive, several research lines emerge targeting to make legacy applications adapt to TEE at a lower cost. Among these, some studies consider combining techniques such as LibOS [12, 76, 82], Docker [4], Sandbox [40], and micro-container [77] on top of TEE, by which legacy applications are protected against attacks from the execution platform or other platform users. In the meanwhile, delicate implementations for specific application areas are proposed, including search index [52, 65], secure storage [7, 42, 80], distributed lease [81], networking [63], machine learning [57], and query processing [1, 2, 8, 43, 83, 90]. In the realm of data analytics, there are solutions designed to safeguard against malicious computing platforms. These solutions involve managing distributed tasks using frameworks such as MapReduce [27, 56, 73] and Spark [64, 102].

TEE-integrated Language Environment. TEEs require developers to devise customized "artifacts" for their applications, adhering to specific programming models or container abstractions, adding engineering complexity. To address this, several studies have suggested the use of higher-level language abstractions, enabling compilers or runtimes to manage TEE and non-TEE hardware during the execution of source code. For example, Uranus [41] implements a Java solution that operates two separate JVMs both inside and outside of the TEE, using annotations to designate the execution environment for functions. GOTEE [34] capitalizes on the unique features of the Go language, promoting communication and execution between independent Go runtimes within and outside of the TEE on a per-goroutine basis. In a similar vein, MesaPy [89] enhances the Python language. These studies operate under the assumption that both the source code and its compilers/interpreters of the "high-level language" are trustworthy, as are the pre-compilation facilities that generate enclaves, typically unprotected by TEE. At their core, these strategies simplify the tasks of trusted coordinators when leveraging untrusted computational resources.

TEE-based Bilateral Barrier. The original purpose of TEE is to create a unidirectional barrier that could house sensitive programs, rendering execution within the barrier inaccessible to the outside world; however, certain studies suggest the construction of a bilateral barrier, where programs executed within the TEE also cannot cross over and impact the outside. Examples include Ryoan [40] which adapts NaCl sandbox [98] within the enclave, and VC3 [73] which employs CFI-style memory isolation by a variant of C# compiler, restricting the enclave's potential impact on the outside. In broad terms, TEE-based bilateral barriers share similarities with **TGCB**, but the selected technical solution results in a large and complex TCB, making it challenging for most data owners to verify the trustworthiness of their code. Instead, they have to trust that open-source implementations from reputable developers are benign, increasing the possibility of diverse vulnerabilities. Furthermore, their threat models should not include "Denial-of-Service" attacks. While programs within the

enclave have limited ability to collaborate with external entities, the underlying computations remain beyond the control of the data owner.

Privacy-preserved Data Federation. Data federation enables the management of multiple data sources in a way that allows each source to operate independently, yet still support joint computation. Unlike traditional distributed databases, this does not necessitate direct sharing or exposure, thereby ensuring the confidentiality of each data source. To uphold this confidentiality, much of the associated research relies on the *Secure Multi-party Computation* (SMC) framework [101]. This either presupposes the existence of an honest broker to convert the joint analysis task into relevant SMC functions [9], or requires that all participating parties strictly adhere to predetermined protocols [86], known as the semi-honest adversary. However, it is clear that the decentralization and security benefits of data federation are often accompanied by significant performance costs. This prompts a series of research initiatives with the aim of optimizing performance for specific use cases. For example, Hu-Fu [60] is designed for spatial data analysis; ShrinkWrap [10] incorporates differential privacy to reveal more intermediate data; SAQE [11] boosts overall performance through the use of approximate query processing. **TGCB** differs from these studies as it does not make assumptions about the data intake. Nevertheless, integrating **TGCB** with these methods holds promise and feasibility when a complex data source is being deployed. As a TEE-based computational backend, **TGCB**'s behavior is restricted, weakening the data federation's semi-honest adversary assumption to meet stronger threat models.

Data Escrow. The primary challenge in privacy-preserving data analysis is the irreversible nature of data disclosure. **TGCB** stands from the perspective of the data owner to ensure that the data remain confidential during SDDP. In practical scenarios, data consumers encounter a similar challenge of assessing the quality of a particular data source to meet specific analysis tasks while respecting its privacy, i.e., the negotiation process preceding data consumption. To tackle this issue, DataStation [96] suggests achieving "Data Escrow" by leveraging TEEs and encryption technologies. Due to centralized data management, both "negotiation" and subsequent "analysis" can be performed ***locally*** and ***in plaintext***, resulting in a significant performance improvement. Specifically, DataStation focuses on trusted data flow control and data quality inspection, requiring that the "functions" agreed upon by both the data owner and the consumer be provided in advance. Despite sharing similar technology stacks, **TGCB** accomplishes the trust shift in delegated computation by proposing EPL, eliminating the need for pre-setting functions, and ensuring ownership transitivity of outputs. However, **TGCB**, as a computational backend, does not take into account the specific methods of storage, sharing, and transmission of encrypted data. Therefore, DataStation and **TGCB** are complementary, meaning that DataStation can serve as a specific implementation of the computational flow model (Section 4.2), and the EPL interpreter can enhance the generality of delegated computation in the data-sharing consortia that DataStation focuses on.

9 CONCLUSION

We propose **TGCB** as a means of supporting SDDP for arbitrary functions without requiring prior inspection and ensuring data confidentiality, aided by its capability of executing EPL scripts in TEE. As a suite of concise, comprehensible, and stable trusted code, once verified reliable, **TGCB** is able to provide long-lasting services as the computational backend. Through a case study in which **TGCB** functions as the computational backend of Spark, we demonstrate the benefits that **TGCB** brings to data confidentiality in general-purpose computational tasks, which renders it more universal and attainable to achieve SDDP from a practical engineering perspective.

REFERENCES

[1] 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. 2020. Azure SQL Database Always Encrypted. In *SIGMOD Conference*. ACM, 1511–1525.

[2] Arvind Arasu, Ken Eguro, Manas Joglekar, Raghav Kaushik, Donald Kossmann, and Ravi Ramamurthy. 2015. Transaction processing on confidential data using cipherbase. In *ICDE*. IEEE Computer Society, 435–446.

[3] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *SIGMOD Conference*. ACM, 1383–1394.

[4] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, André Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Peter R. Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *OSDI*. USENIX Association, 689–703.

[5] Mikhail J. Atallah and Keith B. Frikken. 2010. Securely outsourcing linear algebra computations. In *AsiaCCS*. ACM, 48–59.

[6] Ahmad Al Badawi, Jack Bates, Flávio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Ian Quah, Yuriy Polyakov, R. V. Saraswathy, Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. 2022. OpenFHE: Open-Source Fully Homomorphic Encryption Library. In *WAHC@CCS*. ACM, 53–63.

[7] Maurice Bailleu, Dimitra Giantsidi, Vasilis Gavrielatos, Do Le Quoc, Vijay Nagarajan, and Pramod Bhatotia. 2021. Avocado: A Secure In-Memory Distributed Storage System. In *USENIX Annual Technical Conference*. USENIX Association, 65–79.

[8] Sumeet Bajaj and Radu Sion. 2011. TrustedDB: a trusted hardware based database with privacy and data confidentiality. In *SIGMOD Conference*. ACM, 205–216.

[9] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel N. Kho, and Jennie Rogers. 2017. SMCQL: Secure Query Processing for Private Data Networks. *Proc. VLDB Endow.* 10, 6 (2017), 673–684.

[10] Johes Bater, Xi He, William Ehrich, Ashwin Machanavajjhala, and Jennie Rogers. 2018. ShrinkWrap: Efficient SQL Query Processing in Differentially Private Data Federations. *Proc. VLDB Endow.* 12, 3 (2018), 307–320.

[11] Johes Bater, Yongjoo Park, Xi He, Xiao Wang, and Jennie Rogers. 2020. SAQE: Practical Privacy-Preserving Approximate Query Processing for Data Federations. *Proc. VLDB Endow.* 13, 11 (2020), 2691–2705.

[12] Andrew Baumann, Marcus Peinado, and Galen C. Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven. In *OSDI*. USENIX Association, 267–283.

[13] David Benjamin and Mikhail J. Atallah. 2008. Private and Cheating-Free Outsourcing of Algebraic Computations. In *PST*. IEEE Computer Society, 240–245.

[14] Paul Bernays. 1936. Alonzo Church. An unsolvable problem of elementary number theory. American journal of mathematics, vol. 58 (1936), pp. 345–363. *The Journal of Symbolic Logic* 1, 2 (1936), 73–74.

[15] Marina Blanton, Mikhail J. Atallah, Keith B. Frikken, and Qutaibah M. Malluhi. 2012. Secure and Efficient Outsourcing of Sequence Comparisons. In *ESORICS (Lecture Notes in Computer Science, Vol. 7459)*. Springer, 505–522.

[16] Corrado Böhm and Giuseppe Jacopini. 1966. Flow diagrams, turing machines and languages with only two formation rules. *Commun. ACM* 9, 5 (1966), 366–371.

[17] Zvika Brakerski and Vinod Vaikuntanathan. 2011. Efficient Fully Homomorphic Encryption from (Standard) LWE. In *FOCS*. IEEE Computer Society, 97–106.

[18] Longbing Cao. 2017. Data Science: A Comprehensive Overview. *ACM Comput. Surv.* 50, 3 (2017), 43:1–43:42.

[19] Fei Chen, Tao Xiang, and Yuanyuan Yang. 2014. Privacy-preserving and verifiable protocols for scientific computation outsourcing to the cloud. *J. Parallel Distributed Comput.* 74, 3 (2014), 2141–2151.

[20] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. 2018. Labeled PSI from Fully Homomorphic Encryption with Malicious Security. In *CCS*. ACM, 1223–1237.

[21] Long Cheng, Fang Liu, and Danfeng Yao. 2017. Enterprise data breach: causes, challenges, prevention, and future directions. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 7, 5 (2017), e1211.

[22] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: Fast Fully Homomorphic Encryption Over the Torus. *J. Cryptol.* 33, 1 (2020), 34–91.

[23] Alan Clements. 2013. *Predication.* Computer Organization & Architecture: Themes and Variations, Cengage Learning, Chapter 8.3.7, 532–539.

[24] Jean-Sébastien Coron, Tancrède Lepoint, and Mehdi Tibouchi. 2013. Practical Multilinear Maps over the Integers. In *CRYPTO (1) (Lecture Notes in Computer Science, Vol. 8042)*. Springer, 476–493.

[25] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptol. ePrint Arch.* (2016), 86.

[26] Vasant Dhar. 2013. Data science and prediction. *Commun. ACM* 56, 12 (2013), 64–73.

[27] Tien Tuan Anh Dinh, Prateek Saxena, Ee-Chien Chang, Beng Chin Ooi, and Chunwang Zhang. 2015. M2R: Enabling Stronger Privacy in MapReduce Computation. In *USENIX Security Symposium*. USENIX Association, 447–462.
[28] Changyu Dong, Liqun Chen, and Zikai Wen. 2013. When private set intersection meets big data: an efficient and scalable protocol. In *CCS*. ACM, 789–800.
[29] Muhammad El-Hindi, Carsten Binnig, Arvind Arasu, Donald Kossmann, and Ravi Ramamurthy. 2019. BlockchainDB - A Shared Database on Blockchains. *Proc. VLDB Endow.* 12, 11 (2019), 1597–1609.
[30] Shufan Fei, Zheng Yan, Wenxiu Ding, and Haomeng Xie. 2022. Security Vulnerabilities of SGX and Countermeasures: A Survey. *ACM Comput. Surv.* 54, 6 (2022), 126:1–126:36.
[31] Johannes Gehrke, Lindsay Allen, Panagiotis Antonopoulos, Arvind Arasu, Joachim Hammer, James Hunter, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, Srinath T. V. Setty, Jakub Szymaszek, Alexander van Renen, Jonathan Lee, and Ramarathnam Venkatesan. 2019. Veritas: Shared Verifiable Databases and Tables in the Cloud. CIDR.
[32] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *STOC*. ACM, 169–178.
[33] Craig Gentry, Shai Halevi, and Nigel P. Smart. 2012. Homomorphic Evaluation of the AES Circuit. In *CRYPTO (Lecture Notes in Computer Science, Vol. 7417)*. Springer, 850–867.
[34] Adrien Ghosn, James R. Larus, and Edouard Bugnion. 2019. Secured Routines: Language-based Construction of Trusted Execution Environments. In *USENIX Annual Technical Conference*. USENIX Association, 571–586.
[35] Eric Goldman. 2020. An introduction to the california consumer privacy act (CCPA). *Santa Clara Univ. Legal Studies Research Paper* (2020).
[36] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *STOC*. ACM, 218–229.
[37] Shafi Goldwasser and Silvio Micali. 1982. Probabilistic Encryption and How to Play Mental Poker Keeping Secret All Partial Information. In *STOC*. ACM, 365–377.
[38] Shruthi Gorantala, Rob Springer, Sean Purser-Haskell, William Lam, Royce J. Wilson, Asra Ali, Eric P. Astor, Itai Zukerman, Sam Ruth, Christoph Dibak, Phillipp Schoppmann, Sasha Kulankhina, Alain Forget, David Marn, Cameron Tew, Rafael Misoczki, Bernat Guillen, Xinyu Ye, Dennis Kraft, Damien Desfontaines, Aishe Krishnamurthy, Miguel Guevara, Irippuge Milinda Perera, Yurii Sushko, and Bryant Gipson. 2021. A General Purpose Transpiler for Fully Homomorphic Encryption. *IACR Cryptol. ePrint Arch.* (2021), 811.
[39] Eric Horvitz and Deirdre Mulligan. 2015. Data, privacy, and the greater good. *Science* 349, 6245 (2015), 253–255.
[40] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. 2016. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *OSDI*. USENIX Association, 533–549.
[41] Jianyu Jiang, Xusheng Chen, Tsz On Li, Cheng Wang, Tianxiang Shen, Shixiong Zhao, Heming Cui, Cho-Li Wang, and Fengwei Zhang. 2020. Uranus: Simple, Efficient SGX Programming and its Applications. In *AsiaCCS*. ACM, 826–840.
[42] Robert Krahn, Bohdan Trach, Anjo Vahldiek-Oberwagner, Thomas Knauth, Pramod Bhatotia, and Christof Fetzer. 2018. Pesos: policy enhanced secure object store. In *EuroSys*. ACM, 25:1–25:17.
[43] Mingyu Li, Xuyang Zhao, Le Chen, Cheng Tan, Huorong Li, Sheng Wang, Zeyu Mi, Yubin Xia, Feifei Li, and Haibo Chen. 2023. Encrypted Databases Made Secure Yet Maintainable. In *OSDI*. USENIX Association, 117–133.
[44] Keng-Pei Lin and Ming-Syan Chen. 2010. Privacy-preserving outsourcing support vector machines with random transformation. In *KDD*. ACM, 363–372.
[45] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. 2010. On Ideal Lattices and Learning with Errors over Rings. In *EUROCRYPT (Lecture Notes in Computer Science, Vol. 6110)*. Springer, 1–23.
[46] J MacQueen. 1967. Classification and analysis of multivariate observations. In *5th Berkeley Symp. Math. Statist. Probability*. 281–297.
[47] John McCarthy. 1960. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM* 3, 4 (1960), 184–195.
[48] Brendan McQuade, Lorax B Horne, Zach Wehrwein, and Milo Z Trujillo. 2022. The secret of BlueLeaks: security, police, and the continuum of pacification. *Small Wars & Insurgencies* 33, 4-5 (2022), 693–719.
[49] Catherine A. Meadows. 1986. A More Efficient Cryptographic Matchmaking Protocol for Use in the Absence of a Continuously Available Third Party. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 134–137.
[50] Nicholas Metropolis and Stanislaw Ulam. 1949. The monte carlo method. *Journal of the American statistical association* 44, 247 (1949), 335–341.
[51] Gary L. Miller. 1976. Riemann's Hypothesis and Tests for Primality. *J. Comput. Syst. Sci.* 13, 3 (1976), 300–317.
[52] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. 2018. Oblix: An Efficient Oblivious Search Index. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 279–296.
[53] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 19–38.
[54] David Molnar, Matt Piotrowski, David Schultz, and David A. Wagner. 2005. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *ICISC (Lecture Notes in Computer Science,*

Vol. 3935). Springer, 156–168.
[55] Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. 2013. Privacy-Preserving Ridge Regression on Hundreds of Millions of Records. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 334–348.
[56] Olga Ohrimenko, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Markulf Kohlweiss, and Divya Sharma. 2015. Observing and Preventing Leakage in MapReduce. In *CCS*. ACM, 1570–1581.
[57] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious Multi-Party Machine Learning on Trusted Processors. In *USENIX Security Symposium*. USENIX Association, 619–636.
[58] Pascal Paillier. 1999. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *EUROCRYPT (Lecture Notes in Computer Science, Vol. 1592)*. Springer, 223–238.
[59] Marie Paindavoine and Bastien Vialla. 2015. Minimizing the Number of Bootstrappings in Fully Homomorphic Encryption. In *SAC (Lecture Notes in Computer Science, Vol. 9566)*. Springer, 25–43.
[60] Xuchen Pan, Yongxin Tong, Chunbo Xue, Zimu Zhou, Junping Du, Yuxiang Zeng, Yexuan Shi, Xiaofei Zhang, Lei Chen, Yi Xu, Ke Xu, and Weifeng Lv. 2022. Hu-Fu: A Data Federation System for Secure Spatial Queries. *Proc. VLDB Endow.* 15, 12 (2022), 3582–3585.
[61] Linnea Passing, Manuel Then, Nina C. Hubig, Harald Lang, Michael Schreier, Stephan Günnemann, Alfons Kemper, and Thomas Neumann. 2017. SQL- and Operator-centric Data Analytics in Relational Main-Memory Databases. In *EDBT*. OpenProceedings.org, 84–95.
[62] Japan Personal Information Protection Commission. 2020. Amended Act on the Protection of Personal Information.
[63] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. 2018. SafeBricks: Shielding Network Functions in the Cloud. In *NSDI*. USENIX Association, 201–216.
[64] Do Le Quoc, Franz Gregor, Jatinder Singh, and Christof Fetzer. 2019. SGX-PySpark: Secure Distributed Data Analytics. In *WWW*. ACM, 3564–3563.
[65] Kui Ren, Yu Guo, Jiaqi Li, Xiaohua Jia, Cong Wang, Yajin Zhou, Sheng Wang, Ning Cao, and Feifei Li. 2020. HybrIDX: New Hybrid Index for Volume-hiding Range Queries in Data Outsourcing Services. In *ICDCS*. IEEE, 23–33.
[66] Xuanle Ren, Le Su, Zhen Gu, Sheng Wang, Feifei Li, Yuan Xie, Song Bian, Chao Li, and Fan Zhang. 2022. HEDA: Multi-Attribute Unbounded Aggregation over Homomorphically Encrypted Database. *Proc. VLDB Endow.* 16, 4 (2022), 601–614.
[67] Peter Rindal and Mike Rosulek. 2016. Faster Malicious 2-Party Secure Computation with Online/Offline Dual Execution. In *USENIX Security Symposium*. USENIX Association, 297–314.
[68] Peter Rindal and Mike Rosulek. 2017. Improved Private Set Intersection Against Malicious Adversaries. In *EUROCRYPT (1) (Lecture Notes in Computer Science, Vol. 10210)*. 235–259.
[69] Ronald L Rivest, Len Adleman, Michael L Dertouzos, et al. 1978. On data banks and privacy homomorphisms. *Foundations of secure computation* 4, 11 (1978), 169–180.
[70] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. 1978. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM* 21, 2 (1978), 120–126.
[71] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. 2015. Trusted Execution Environment: What It is, and What It is Not. In *TrustCom/BigDataSE/ISPA (1)*. IEEE, 57–64.
[72] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. 2018. ZeroTrace : Oblivious Memory Primitives from Intel SGX. In *NDSS*. The Internet Society.
[73] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 38–54.
[74] Imtiyazuddin Shaik, Nishanth Chandran, et al. 2022. Privacy and data protection in the enterprise world. *CSI Transactions on ICT* 10, 1 (2022), 37–45.
[75] Zihao Shan, Kui Ren, Marina Blanton, and Cong Wang. 2018. Practical Secure Computation Outsourcing: A Survey. *ACM Comput. Surv.* 51, 2 (2018), 31:1–31:40.
[76] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 2020. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *ASPLOS*. ACM, 955–970.
[77] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. 2017. Panoply: Low-TCB Linux Applications With SGX Enclaves. In *NDSS*. The Internet Society.
[78] Xiaokui Shu, Ke Tian, Andrew Ciambrone, and Danfeng Yao. 2017. Breaking the Target: An Analysis of Target Data Breach and Lessons Learned. *CoRR* abs/1701.04940 (2017).
[79] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: an extremely simple oblivious RAM protocol. In *CCS*. ACM, 299–310.

[80] Yuanyuan Sun, Sheng Wang, Huorong Li, and Feifei Li. 2021. Building Enclave-Native Storage Engines for Practical Encrypted Databases. *Proc. VLDB Endow.* 14, 6 (2021), 1019–1032.
[81] Bohdan Trach, Rasha Faqeh, Oleksii Oleksenko, Wojciech Ozga, Pramod Bhatotia, and Christof Fetzer. 2020. T-Lease: a trusted lease primitive for distributed systems. In *SoCC*. ACM, 387–400.
[82] Chia-che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *USENIX Annual Technical Conference*. USENIX Association, 645–658.
[83] Harshavardhan Unnibhavi, David Cerdeira, Antonio Barbalace, Nuno Santos, and Pramod Bhatotia. 2022. Secure and Policy-Compliant Query Processing on Heterogeneous Computational Storage Architectures. In *SIGMOD Conference*. ACM, 1462–1477.
[84] Cor J. Veenman, Marcel J. T. Reinders, and Eric Backer. 2002. A Maximum Variance Cluster Algorithm. *IEEE Trans. Pattern Anal. Mach. Intell.* 24, 9 (2002), 1273–1280.
[85] Paul Voigt and Axel Von dem Bussche. 2017. The EU general data protection regulation (GDPR). *A Practical Guide, 1st Ed., Cham: Springer International Publishing* 10, 3152676 (2017), 10–5555.
[86] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. 2019. Conclave: secure multi-party computation on big data. In *EuroSys*. ACM, 3:1–3:18.
[87] Cong Wang, Kui Ren, and Jia Wang. 2011. Secure and practical outsourcing of linear programming in cloud computing. In *INFOCOM*. IEEE, 820–828.
[88] Cong Wang, Kui Ren, Jia Wang, and Qian Wang. 2013. Harnessing the Cloud for Securely Outsourcing Large-Scale Systems of Linear Equations. *IEEE Trans. Parallel Distributed Syst.* 24, 6 (2013), 1172–1181.
[89] Huibo Wang, Mingshen Sun, Qian Feng, Pei Wang, Tongxin Li, and Yu Ding. 2020. Towards Memory Safe Python Enclave for Security Sensitive Computation. *CoRR* abs/2005.05996 (2020).
[90] 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. 2022. Operon: An Encrypted Database for Ownership-Preserving Data Management. *Proc. VLDB Endow.* 15, 12 (2022), 3332–3345.
[91] Wei Wang, Yin Hu, Lianmu Chen, Xinming Huang, and Berk Sunar. 2012. Accelerating fully homomorphic encryption using GPU. In *HPEC*. IEEE, 1–5.
[92] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. 2015. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *CCS*. ACM, 850–861.
[93] Xiao Shaun Wang, Yan Huang, Yongan Zhao, Haixu Tang, XiaoFeng Wang, and Diyue Bu. 2015. Efficient Genome-Wide, Privacy-Preserving Similar Patient Query based on Private Edit Distance. In *CCS*. ACM, 492–503.
[94] Jaap Wieringa, PK Kannan, Xiao Ma, Thomas Reutterer, Hans Risselada, and Bernd Skiera. 2021. Data analytics in a privacy-concerned world. *Journal of Business Research* 122 (2021), 915–925.
[95] Wai Kit Wong, David Wai-Lok Cheung, Ben Kao, and Nikos Mamoulis. 2009. Secure kNN computation on encrypted databases. In *SIGMOD Conference*. ACM, 139–152.
[96] Siyuan Xia, Zhiru Zhu, Chris Zhu, Jinjin Zhao, Kyle Chard, Aaron J. Elmore, Ian T. Foster, Michael J. Franklin, Sanjay Krishnan, and Raul Castro Fernandez. 2022. Data Station: Delegated, Trustworthy, and Auditable Computation to Enable Data-Sharing Consortia with a Data Escrow. *Proc. VLDB Endow.* 15, 11 (2022), 3172–3185.
[97] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets (Extended Abstract). In *FOCS*. IEEE Computer Society, 162–167.
[98] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 79–93.
[99] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*. USENIX Association, 15–28.
[100] Yihua Zhang and Marina Blanton. 2014. Efficient Secure and Verifiable Outsourcing of Matrix Multiplications. In *ISC (Lecture Notes in Computer Science, Vol. 8783)*. Springer, 158–178.
[101] Chuan Zhao, Shengnan Zhao, Minghao Zhao, Zhenxiang Chen, Chong-Zhi Gao, Hongwei Li, and Yu-an Tan. 2019. Secure Multi-Party Computation: Theory, practice and applications. *Inf. Sci.* 476 (2019), 357–372.
[102] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *NSDI*. USENIX Association, 283–298.
[103] Lifeng Zhou and Chunguang Li. 2016. Outsourcing Eigen-Decomposition and Singular Value Decomposition of Large Matrix to a Public Cloud. *IEEE Access* 4 (2016), 869–879.
[104] Wenchao Zhou, Yifan Cai, Yanqing Peng, Sheng Wang, Ke Ma, and Feifei Li. 2021. VeriDB: An SGX-based Verifiable Database. In *SIGMOD Conference*. ACM, 2182–2194.

Received April 2023; revised July 2023; accepted August 2023