Confidential Analytics with Scylla

Shamiek Mangipudi
Università della Svizzera italiana (USI)
Switzerland
shamiekm@gmail.com

Pavel Chuprikov
Télécom Paris, Institut Polytechnique de Paris
France
pavel.chuprikov@telecom-paris.fr

Gerald Prendi
Università della Svizzera italiana (USI)
Switzerland
geraldprendi18@gmail.com

Patrick Eugster
Università della Svizzera italiana (USI)
Switzerland
eugstp@usi.ch

ABSTRACT

While security concerns of data at rest and in transit have been addressed over the years using standard cryptographic measures, those surrounding data in use have garnered significant attention in recent times. In response, various *trusted execution environment*s (TEEs) have been proposed and are on offer from leading public cloud providers. With development and re-programming efforts, availability, threat models, pricing, performance, etc., differing between various TEEs themselves and also with viable alternatives such as software solutions like *partially homomorphic encryption* (PHE) to protect data in use, it is imperative to have a system that is independent of these several varying dimensions while also efficiently achieving end-to-end confidentiality guarantees on data processing.

We propose Scylla, a mechanism-agnostic confidential analytics framework, built on top of the popular Spark data analytics engine. Scylla utilizes a customizable combination of TEEs and PHE schemes to achieve end-to-end confidentiality guarantees with prime performance. Our evaluation shows that Scylla's query execution times are 1.91× faster than state-of-the-art system Opaque providing similar guarantees. Scylla's novel general architecture enables integrating latest TEEs such as AWS Nitro, AMD SEV-SNP, and Intel TDX with zero rebuilding efforts.

CCS CONCEPTS

• **Security and privacy → Distributed systems security**.

KEYWORDS

Confidential Computing, Nitro enclave, TDX, SEV-SNP, SGX, PHE

ACM Reference Format:
Shamiek Mangipudi, Pavel Chuprikov, Gerald Prendi, and Patrick Eugster. 2025. Confidential Analytics with Scylla. In *ACM Symposium on Cloud Computing (SoCC '25), November 19–21, 2025, Online, .* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3772052.3772209

1 INTRODUCTION

Data analytics – as a means of learning from large data sets – continues to gain in importance across many domains including business, finance, governance, healthcare, and science. To deal with increasingly large data sets in a cost-efficient manner, data analytics frameworks are commonly deployed in cloud infrastructures. Meanwhile, according to a 2024 study by IBM, 80% of data breaches happen in the cloud [55], exacerbating concerns over security of data *in use* for cloud-based data analytics.

Many *security mechanisms* (in short, *mechanisms*) have been proposed and promoted over the years for protecting data in use particularly in cloud infrastructures, including *software* and *hardware* mechanisms. Examples for software mechanisms include cryptographic system*s* (in short, schemes) for *homomorphic encryption* (HE), such as PHE through schemes like ElGamal [38], Paillier [84], or *symmetric multiplicative homomorphic encryption* (SMHE) and *symmetric additive homomorphic encryption* (SAHE) [92]. For hardware mechanisms, major processor vendors have introduced *trusted execution environment*s (TEEs) to the market, e.g., Intel *software guard extensions* (SGX), Intel *trust domains extensions* (TDX), AMD *secure encrypted virtualization-secure nested paging* (SEV-SNP). Recently, Amazon has added its own hardware mechanism AWS *Nitro* to EC2. Many approaches protecting big data processing use access control and fall short due to lack of usage of cryptographic measures [106, 107, 113, 114] although Airavat [90] adds mathematically rigorous measures like differential privacy, while others are built around a fixed design point, i.e., a specific mechanism such as Intel SGX [8, 17, 65, 88, 94, 96, 97, 103, 120], thus lacking generality, and have to be redesigned for every new mechanism rolled out.

Confidential Analytics. While software and hardware mechanisms offer possibilities for securing scalable data analytics frameworks, their efficient usage presents several challenges, which we collectively dub the site challenges.

Security: Different mechanisms – even just considering TEEs – provide quite different security guarantees [119, Table 1] which are often not (yet) rigorously specified. For instance, AWS Nitro isolates enclaves using its custom Nitro system, which combines a lightweight hypervisor, dedicated Nitro hardware and software, and CPU hardware virtualization. While Nitro also uses memory encryption, its enclave isolation relies on this broader hardware and virtualization architecture, differing from the *virtual machine* (VM)-centric memory encryption of SEV-SNP and TDX. Furthermore, TEEs are subject to different threats and attacks (e.g., [23, 63, 117] for SGX, SEV-SNP, TrustZone respectively).

Independence: Major clouds (AWS, Azure, GCP) support different sets of TEEs while new TEEs are in the making. Moreover, while hardware mechanisms are expected to become yet more

Work partially supported by SNSF Bridge PoC #40B1-0_233182, ANR-22-EXES-0013 (project STeP2/F4C), Cisco Research University Funding #2853380, Hasler Foundation #20053, and Meta Security Research #474960397718052.

SoCC '25, November 19–21, 2025, Online, USA

ACM ISBN 979-8-4007-2276-9/2025/11.
https://doi.org/10.1145/3772052.3772209

efficient, some software mechanisms like certain PHE schemes are still competitive in terms of performance in distributed setups [71], and also can deal with worries of internal threats at cloud providers and hardware vendors and government-required back doors. Besides, with mechanisms not uniformly available, requiring different programmability efforts, possessing different development interfaces, it is vital to have a framework which is independent of these variables, so as to save system rebuilding/-porting efforts for every upcoming mechanism.

Transparency: Data analysts are typically not expert programmers, let alone security experts, and cannot be expected to manually program with security mechanisms. Meanwhile, even just within the class of TEEs, various solutions possess distinct microarchitectural features, hardware interfaces, functionalities, underlying hardware architectures [75] which adds to the difficulty of using them.

Efficiency: As data analytics relies on ever-larger datasets, differences in performance overhead of mechanisms [47, 75, 76], even though seemingly small, become substantial at scale. This substantial overhead difference directly leads to significantly increased query execution times and higher cloud costs, a deterrent for many potential users who may then forgo security measures or avoid cloud adoption altogether, thereby losing out on significant opportunities.

The need for T and E has in the past motivated various approaches specifically for confidential data analytics (e.g., [85, 93, 120]) beyond platforms for cloud security focusing on more general computing and programming models (e.g., [1, 110]), yet the quest for a solution addressing all four SITE requirements in a satisfactory manner remains elusive.

Scylla. We propose Scylla, a novel solution for efficient confidential analytics that is transparent to users while being secure yet platform-independent. More specifically, our system design is security-centered on the recently proposed guarantee of *generalized policy-based non-interference* [71] that allows for the integration of different mechanisms by means of a novel extended security policy (S), while being independent of any specific mechanism (I). Our design introduces a secure runtime, and a query processing pipeline for automated transformation and verification of queries expressed by users in a security-agnostic manner (T). The pipeline includes a framework for easily plugging in new software and hardware mechanisms. This framework comprises APIs as well as a *domain-specific language* (DSL) to express heuristics for using arbitrary combinations of mechanisms as part of automated query transformation. The ability to combine different mechanisms supports optimal performance (E), besides portability and interoperability.

Contributions. In short this paper's key contributions are:

- A novel approach with a general architecture for end-to-end confidential analytics, independent of software and hardware mechanisms.
- Unified abstractions – capturing differences between TEEs and PHEs schemes, respective guarantees, performance characteristics, availability in cloud providers – decoupling three aspects: security policy, mechanism definition/integration, and heuristics for automated mechanism use.
- A framework for plugging in mechanisms (supporting mechanisms at different abstraction levels, e.g., `NativeTEE`, `VirtualTEE`) and a Scala DSL for expressing novel rich transformation heuristics to efficiently employ the mechanisms. Our DSL allows to assign mechanisms in arbitrary orders and combinations.
- An efficient design of our approach and its implementation as extension to Apache Spark (Spark core, Spark standalone cluster manager, and Spark SQL).
- Support for Nitro including a novel communication setup between remote *Nitro enclave*s (NEs) using *transparent socket impersonation* (TSI).
- An evaluation of our system in 2 different clouds – AWS and Azure – using 5 PHE schemes and 4 hardware mechanisms SGX, SEV-SNP, TDX, and Nitro. To the best of our knowledge, this is the first study of Nitro's performance at scale. Scylla adds minor overheads through its general architecture (I), and its ability to combine mechanisms leads to better efficiency (E). On average, when using PHE, Scylla is 1.02× faster than closest related work Hydra [71]. When using SGX, Scylla is 1.91× faster than Opaque [120], and Hydra is 1.19× faster than Scylla. However, neither Opaque nor Hydra support SEV-SNP, TDX, or Nitro. A novel heuristic obtained by changing only a few lines of code further increases Scylla's performance by 1.36×.

Roadmap. § 2 presents background information on the mechanisms supported in Scylla and on Spark which Scylla builds on. § 3 overviews Scylla's architecture and workflow; § 4 presents Scylla's threat model, security policy, core language, and guarantees. § 5 introduces Scylla's API and DSL, and § 6 presents Scylla's runtime design. § 7 evaluates Scylla in comparison with existing systems. § 8 contrasts Scylla with related work. § 9 draws final conclusions.

2 BACKGROUND

This section presents pertinent background on TEEs and HE. We first give a general description of these technologies, and then expand on specific realizations used in Scylla – SGX, TDX, SEV-SNP, Nitro, and PHE – and present performance considerations.

Trusted Execution Environments. A TEE enables isolated execution contexts that enjoy strong protection guarantees – confidentiality and integrity of code and data– from the rest of the system including host OS, hypervisor, firmware etc.

Intel *software guard extensions* (SGX) provides the original abstraction of user-space *enclaves*, isolated execution contexts implemented as secure regions of user-mode address space. Intel SGX protects confidentiality and integrity of pages in an enclave while relying on the untrusted host OS for scheduling, memory-management, I/O, etc. Enclave pages are encrypted and integrity protected by a dedicated on-die component, the *memory encryption engine* (MEE), when written to memory. Decryption occurs transparently on the die upon valid access of an enclave page. The latest version SGXv2 [74, 98, 111] has increased the capacity of the protected memory region to up to 512 GB per socket and reduced overhead of memory protection.

Trust domains extensions (TDX) [56] is the latest architectural extension from Intel providing TEE capabilities. TDX allows the

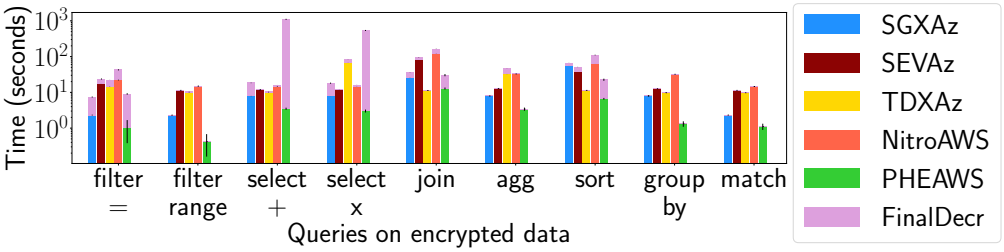

Figure 1: Microbenchmarks on encrypted data (AES-GCM [73] for SGX, *secure encrypted virtualization* (SEV), TDX, Nitro, and compatible schemes for PHE) for various mechanisms. Execution times (log) with final decryption time (as decryption of collected result on the driver/client-side completes the query) stacked on top.

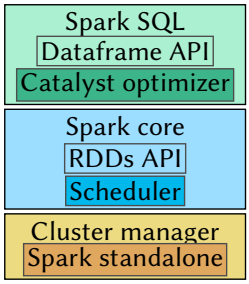

Figure 2: Scylla stack as extension of Spark.

deployment of virtual machines in secure-arbitration mode (SEAM) with encrypted CPU state and memory, integrity protection, and *remote attestation* (RA). This hardware-assisted isolation for VMs, termed trust domains (TDs), minimizes the attack surface exposed to a potentially untrustworthy hypervisor or host *operating system* (OS).

AMD *secure encrypted virtualization* (SEV) [4] uses guest specific encryption keys to isolate entire VMs as kernel-space enclaves which ensures confidentiality and integrity of VMs while relying on the untrusted hypervisor for memory-management, I/O, etc. SEV's latest variant SEV-SNP [3] builds upon SEV and SEV-ES [57] while introducing new hardware-based security protection, e.g., strong memory integrity protection to prevent malicious hypervisor-based attacks that rely on guest data corruption, aliasing, replay, and various other attack vectors. SEV-SNP (hereon called SEV) has additional optional security enhancements which offer stronger protection around interrupt behavior and certain side channel attacks.

The AWS Nitro system [11, 13] enables creation of isolated environments called *Nitro enclaves* (NEs) [9, 10, 12] to protect and process highly trusted code and highly sensitive data. NEs offer an isolated, hardened, and highly constrained environment to host security-critical applications. The Nitro system [13] – the backbone of NEs – carves out resources from the parent instance to create another fully protected independent VM environment to launch an enclave. The Nitro system is built using secure, encrypted, authenticated microservices; it is itself isolated and operates in a substrate network, with no internet or general-purpose access. The Nitro system consists of three components: 1. a series of Nitro cards, 2. a Nitro security chip, and 3. a Nitro hypervisor. 1. includes cards for *virtual private cloud* (VPC), *elastic block store* (EBS), and local *nonvolatile memory express* (NVMe) storage, and the Nitro controller. The Nitro controller is the primary card which along with its secure boot process provides the hardware root of trust in a Nitro system. 2. provides a simple hardware-based root of trust and extends the Nitro controller chain of trust to the system main board. It is integrated into the motherboard, protecting hardware resources/firmware by intercepting and moderating all operations to them. All write access from the instance to non-volatile storage is blocked in hardware by this security chip. The Nitro system provides near-metal capabilities (nearly indistinguishable as measured at Netflix, overhead less than 1% [16, 49]) by offloading virtualization overhead to dedicated hardware and software through the use of Nitro cards, which further minimizes the attack surface of the hypervisor. Offloading simplifies the overall stack thereby minimizing the *trusted computing base* (TCB). Enclaves include cryptographic attestation to ensure that only authorized code is running, and integrate with the AWS *key management service* (KMS), so that only enclaves can access sensitive material. The Nitro system creates a new VM, attests and runs the enclave image to create a NE. Isolation is enforced in hardware through a combination of Nitro virtualization and well-tested in-CPU hardware virtualization. The parent instance has no access or visibility of the enclave's memory or core. This design is simpler, more robust than memory encryption. No data volume or access patterns are revealed. Enclave memory is never malleable.

Homomorphic Encryption. *Homomorphic encryption* (HE) refers to specialized schemes that allow computations to be performed directly on encrypted values. HE includes *fully homomorphic encryption* (FHE) and PHE. FHE supports arbitrary computations over encrypted data. However, despite significant improvements (e.g. [30]), FHE can exhibit high overhead for complex computations [42–44]. Instead, in this paper we focus on PHE [89]. A scheme is said to be partially homomorphic with respect to certain operations if it enables those operations on encrypted data by altering a given ciphertext or combining a set of ciphertexts to get a new one.

Similar in function to PHE is *property-preserving encryption* (PPE). As the name suggests, the ciphertext of these schemes preserve some properties of the underlying plaintext, allowing some operations to be applied directly on the ciphertext. For example, *order-preserving encryption* (OPE) allows order comparisons such as "$\leq$" and "$\geq$" on ciphertexts. For simplicity and brevity in the following we may collectively refer to PHE and PPE schemes as PHE.

TEE vs. PHE: Performance Trade-Offs. TEEs offer code integrity via RA, while PHE does not. PHE schemes are typically implemented via portable easily inspectable code, while TEEs, even for confidentiality, require respective manufacturers to be trusted, as individual hardware elements are not easily inspectable (even if their designs are).

PHE schemes remain performance-wise also relevant for confidentiality in distributed setups. Consider Fig. 1 showing microbenchmark results on 1 M rows of synthetic data for SQL operators using various mechanisms – PHE and Nitro in AWS; SGX, SEV, and TDX in Azure. As can be seen PHE can be quite efficient. This is because in a distributed setup the operator can execute directly on ciphertext, whereas with TEEs data received (resp. sent) must be decrypted (resp. encrypted) before (resp. after) applying an operator. Including the final decryption latency (see mauve parts stacked on top of bars in the graph) of the result at the end user can flip the end-to-end performance trends for some PHE schemes/operators.

Spark. Apache Spark is a distributed cluster computing framework designed to be fast, scalable, and fault-tolerant [116]. A Spark application runs on top of a pluggable external *cluster manager* which manages resource allocation from a distributed cluster of machines required to run the application. Scylla utilizes a custom cluster manager built on Spark's standalone cluster manager to launch the application.

Spark core primarily consists of the *resilient distributed datasets* (RDDs) API for manipulating a distributed collection of objects partitioned across a cluster, and the driver. Spark SQL runs on top of Spark core and introduces a new *dataframe* abstraction for structured data representing a distributed collection of rows with same schema which boasts a tight integration with full programming languages like Scala. This enables intermixing procedural Spark code written using functional programming constructs and relational code leading to more optimized execution compared to what native Spark API can achieve. Spark SQL, implemented using Catalyst–an extensible query optimizer–enables the addition of new rule-based and cost-based optimizations. Scylla builds on this foundation but significantly modifies Catalyst and Spark core, augmenting their fundamental functionality to efficiently address the site challenges. Fig. 2 shows customized Spark components of Scylla's stack in darker shades (i.e., dark green, dark blue, and dark yellow), and vanilla components in lighter shades.

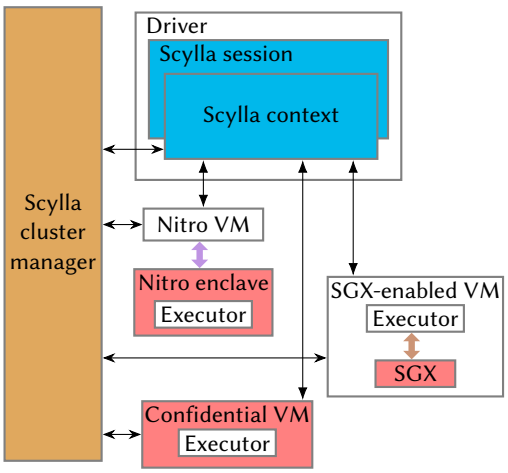

Figure 3: Scylla's general architecture supports enclave-like `NativeTEEs` such as SGX, and VM-like `VirtualTEEs` such as Nitro enclave, SEV, and TDX.

3 DESIGN

This section overviews the design of the Scylla system. Fig. 3 outlines Scylla's runtime architecture.

3.1 Architecture

An external service called Scylla cluster manager (SCM), our custom security-aware cluster manager, is at the core of Scylla's architecture. Inspired by and built on top of Spark's own standalone cluster manager, SCM enables launching executor processes which can be either (a) hosted within VM-like `VirtualTEEs` (e.g., *confidential VMs* (CVMs) like SEV, Nitro, TDX), or (b) have access to native enclave-like `NativeTEEs` (e.g., SGX) hardware mechanisms. SCM can easily be extended with other hardware security mechanisms, for instance, a straightforward *lift and shift* approach would work for a new CVM due to Scylla's general and efficient design. We discuss Scylla's configuration and extension API in § 5.

To cover a broad range of TEEs, Scylla supports a variety of communication channels: vsock [66] to talk to executors hosted in Nitro enclaves, SGX enclaves are accessed by Scylla executors via *Java native interface* (JNI), while executors in other CVMs such as SEV straightaway use the host's network stack. Once a pre-complied Scylla application *Java archive* (JAR) holding analytics queries is submitted to SCM, the Spark driver is deployed on a host within `Client` domain and the application starts executing. The driver runs user code which makes use of `ScyllaSparkSession` that internally interacts with the Spark cluster (represented by `ScyllaSparkContext`). The driver, respecting providers and TEE availability, schedules tasks on executors.

3.2 Workflow

Fig. 4 has Scylla's workflow which spans across Spark SQL and Spark core. Scylla introduces customizations at public extension points within Spark SQL's *query execution* – the primary pipeline for executing relational queries in Spark offering access to intermediate execution phases such as logical planning, optimization, and physical planning. The Spark SQL portion ingests a SQL query as a dataframe constructed using Spark SQL's API. Then along the query execution pipeline, a sequence of transformations and optimizations are applied to the logical and physical plans derived from the SQL query before creating the final *executed plan*, which is handed over to Scylla *directed acyclic graph* (DAG) scheduler in the Spark core portion where a DAG of stages and tasks is created from the executed plan. Scylla task scheduler schedules the tasks corresponding to data partitions on executors. Cryptographic keys are provisioned to the spawned executors once RA is successful. Executors then decrypt → compute → encrypt and return results to the driver where they are collected and decrypted. During execution, data is only ever decrypted either in executors hosted in CVMs (`VirtualTEEs`) or in enclaves (`NativeTEEs`) running alongside executors, always in accordance with security policy $\mathbb{P}$.

4 SECURITY

Scylla's power comes from the ability to utilize an extensible set of hardware (TEEs) and software (PHE schemes) mechanisms, and support different cloud providers or simply providers. A provider represents the internet domain `Inet` or a specific (type of) compute

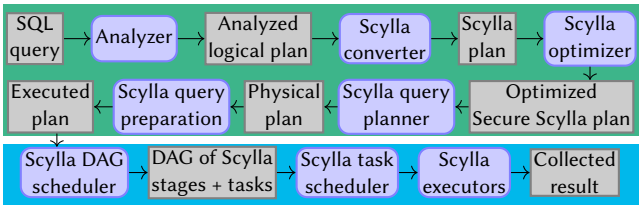

Figure 4: Scylla workflow with Spark SQL and Spark core portions encoded using background colors.

infrastructure where query execution may take place, examples being public clouds (e.g., AWS or Azure), private clouds, but also client domains (Client). The set of hardware mechanisms is denoted as $\mathcal{T}$, the set of schemes as $\mathcal{S}$, and the set of providers as $\mathcal{P}$. We denote all mechanisms as $\mathcal{M} = \mathcal{T} \cup \mathcal{S}$. (Naturally, $\mathcal{T} \cap \mathcal{S} = \emptyset$).

4.1 Threat Model

Mechanisms can provide different guarantees, and be subject to attacks of different types, e.g., side-channel [21, 22, 36, 54, 69, 77, 99, 109, 118], physical [60, 70], fault injection [29, 81, 100], cache [6, 48, 50, 51, 62, 68, 83, 115], controlled-channel [23, 79, 112], and transient-execution [19, 20, 25, 27, 58, 59, 95, 108] attacks. The ease of actually mounting such attacks will depend on many parameters of the cloud infrastructure including the cloud programming abstraction level provided (e.g., platform-as-a-service vs software-as-a-service), the exact hardware and software stack, possibly down to specific versions of components, etc.

To reconcile these constraints Scylla assumes an *honest but curious* (HbC) adversary *at the basis*. Scylla formally guarantees security properties for an HbC adversary, and in general safeguards against the strongest adversary that the weakest mechanism in the system can protect against, with HbC being a safe underapproximation. Protection against adversaries is based on the mechanisms employed as configured through the *security policy* which is determined by the security expert who decides company policies defining which mechanisms and systems to trust for what. Through it's security policy, Scylla allows the security expert to assign/exclude mechanisms appropriately for different security *labels*/*levels* based on strength or different criteria thereof, as currently there is no established comprehensive taxonomy that classifies all attacks according to adversary strength. We do not formally dwell on runtime security and end-to-end guarantees against stronger adversaries. Scylla adopts a complementary approach to works that protect against specialized attack vectors [2, 5] against mechanisms. The security policy can combat yet to be discovered attacks/vulnerabilities, by preventing usage of such afflicted mechanisms, via simple updates to the security policy.

4.2 Security Policy

The security level is thus the basic notion of Scylla's security policy following *multi-level security* (MLS) capturing custom confidentiality requirements for data. We denote the set of possible levels as $\mathcal{L}$. As in previous works, we assume the levels are arranged as a lattice, allowing some (but not all) levels to be compared with "no more secure than", denoted as $l_1 \preccurlyeq l_2$, and any two levels l_1 and l_2 to be merged into $l_1 \sqcup l_2 \in \mathcal{L}$ so that $l_1 \preccurlyeq l_1 \sqcup l_2$ and $l_2 \preccurlyeq l_1 \sqcup l_2$. A data manager, possibly with assistance from the security expert, labels data with respective levels (cf. prior work [78]).

Scylla's security policy $\mathbb{P}$ extends the standard MLS policy, where the latter policy essentially defines *which principal has access to what data*. The extension chiefly includes a mapping that associates with every security level of $\mathcal{L}$ a set of mechanism-provider pairs that are deemed sufficient for protecting data at that level, thus defining *how* data is to be accessed (when/where accessible), and is used by Scylla to guide query execution. Scylla thus streamlines the security expert's job by capturing relevant policies as an artifact, used by the system directly, based on the paradigm of security-policy-as-code [31] (which is similar in spirit to infrastructure-as-code [80], elasticity programming [91], or intent-based networking [40]). The mapping of $\mathbb{P}$ can be easily updated any time after deployment, with immediate application for any subsequently executed queries.

More formally, $\mathbb{P}$ is a function from $\mathcal{L}$ into a subset of $\mathcal{P} \times \mathcal{M}_\perp$, where $\mathcal{M}_\perp = \mathcal{M} \cup \{\perp\}$ with $\perp$ signifying "no protection" (plaintext data processing). List. 1 details the YAML-based configuration language to be used by a security expert for specifying the security policy. The **`lattice`** defines the set of security labels and their relation to each other, while **`policy`**'s **`constraints`** define for each label and provider the set of required security mechanisms, which can be either a **`tee`** or a **`scheme`**, which can possibly be `null` denoting plaintext. Finally **`autoclosure`** controls whether the constraints should be automatically closed by Scylla according to properties outlined shortly afterwards. List. 1 also shows a simple three level lattice (`Pub` $\preccurlyeq$ `Low` $\preccurlyeq$ `High`) and an example security policy – both used later on for evaluation in § 7. As shown in List. 1, for `High` security level we have $\mathbb{P}(\texttt{High}) = \{\langle \texttt{Azure}, \texttt{SGX}\rangle, \langle \texttt{Azure}, \texttt{AES-GCM}\rangle\}$, where Azure denotes the provider and SGX and AES-GCM the respective hardware mechanism (TEE) and software mechanism (scheme), and that `High` data must either be inside SGX or be encrypted with `AES-GCM`; `Low` data requires a scheme (Paillier in this case); public data represented by $\mathbb{P}(\texttt{Pub}) = \{\langle \texttt{AWS}, \perp\rangle\}$ can be stored as plaintext; and transmission between any providers (cf. Inet provider) of `High` (and thus all) data similarly uses `AES-GCM`, while the Client admits plaintext for `High` (and thus all data). We impose the following natural requirements, via **`autoclosure`**, on $\mathbb{P}$: $(p, \perp) \in \mathbb{P}(l)$ implies $(p, m) \in \mathbb{P}(l)$ for all $m \in \mathcal{M}$, and $\mathbb{P}(l) \subseteq \mathbb{P}(l')$ for all $l' \preccurlyeq l$.

Security levels of $\mathcal{L}$ are then assigned to columns of input relations to specify their confidentiality requirements via a *security-aware relational schemata* ρ, which maps table names to relation types, i.e., $\rho : name \mapsto T\{\overline{f : (d, l)}\}$. Another mapping $\rho^p : name \mapsto p$ specifies for each input relation the provider storing that relation. Via ρ^p Scylla ensures: (a) schemes "at rest" satisfy $\mathbb{P}$, and (b) the transfer of input relations between providers via the internet is also protected according to $\mathbb{P}$.

4.3 Secure Query Language

Scylla's guarantees are based on a language abstracting SQL-like data analytics queries. Fig. 5 presents Scylla's core language that brings together all the elements presented so far, while exposing a security-agnostic interface to data analysts. Types κ cover scalars,

```
/* General definition */
lattice:
  label_name: /*≥*/ [label_name]
policy:
  constraints:
    label_name:
      - provider_name:
          {tee: tee_name} |
          {scheme: scheme_name?}
  autoclosure: bool
/* Example */
lattice:
  {Pub: [], Low: [Pub], High: [Low]}
policy:
  constraints:
    Pub: {AWS: [{scheme: null}]}
    Low: {AWS: [{scheme: Paillier}]}
    High:
      Azure: [{tee: SGX},
              {scheme: AES-GCM} /* preset */]
      Inet: [{scheme: AES-GCM}] # preset
      Client: [{scheme: null}] # preset
  autoclosure: true
```

Listing 1: Security policy definition with example (`sec.yaml`).

records, relations, and functions, all building on top of a set of primitive types d, annotated with security levels $l \in \mathcal{L}$ and schemes $s \in \mathcal{S}_\perp$. Both levels and schemes are automatically determined by Scylla (the former from annotated datasets). Values v, i.e., fully computed expressions, directly correspond to the types, the only exception being a field name f which is only used when selecting a grouping field. Most interesting are the expressions (or queries), which include common constructs such as variables, function calls, or binary operators. In addition, there are two constructs fundamental to analytics: input relation references $\texttt{table}(name)$ and relational operators $\theta_{p,t}(\overline{e})$. The latter are *automatically* annotated by Scylla (see § 5.2) with a provider p and an (optional) TEE where the respective relational operator will be executed. Finally, there are encryption/decryption operations, also introduced automatically by Scylla.

4.4 Guarantees

Scylla's guarantee is a form of *noninterference* (NI) [45] based on the *generalized* policy-based NI notion called $\mathbb{S}$*-noninterference* ($\mathbb{S}$-NI) [71]. $\mathbb{S}$-NI generalizes NI to a custom MLS lattice defined as part of a policy $\mathbb{S}$. Scylla uses a different security policy $\mathbb{P}$ and different core language, so we dub its guarantee correspondingly $\mathbb{P}$*-noninterference* ($\mathbb{P}$-NI). Scylla provides the following guarantee:

Definition 4.1 ($\mathbb{P}$-noninterference $\mathbb{P}$-NI$(e)_{\rho,p}$)**.** Expression e has $\mathbb{P}$*-noninterference property* $\mathbb{P}$-NI$(e)_{\rho,p}$ if and only if, for every l in $\mathcal{L}$, for any two ρ-stores db_1 and db_2, $\mathsf{db}_1 \sim^l_\rho \mathsf{db}_2$, and any two values v_1 and v_2, $e \Downarrow_{\mathsf{db}_1} v_1$ and $e \Downarrow_{\mathsf{db}_2} v_2$, it holds that $v_1 \sim^{p,l} v_2$.

Here, ρ is the relational schemata described earlier, $\mathsf{db}_1 \sim^l_\rho \mathsf{db}_2$ states that two stores db_1 and db_2 differ only in data with security level l, $v_1 \sim^{p,l} v_2$ denotes indistinguishability between values v_1 and v_2 by an adversary with respect to whom l must guarantee

$$
\begin{array}{rl}
\text{TEE} & t ::= \perp \mid \texttt{SGX} \mid \texttt{SEV} \mid \texttt{Nitro} \mid \texttt{TDX} \mid \ldots \\
\text{Scheme} & s ::= \perp \mid \texttt{AES-GCM} \mid \texttt{ElGamal} \mid \texttt{Paillier} \mid \ldots \\
\text{Provider} & p ::= \texttt{Client} \mid \texttt{AWS} \mid \texttt{Azure} \mid \texttt{GCP} \mid \texttt{Inet} \mid \ldots \\
\text{Type} & \kappa ::= (d^s, l) \mid \{\overline{f : (d^s, l)}\} \mid T\{\overline{f : (d^s, l)}\} \mid \overline{\kappa} \to \kappa \\
\text{Prim. data type} & d ::= \texttt{Integer} \mid \texttt{Double} \mid \texttt{String} \mid \texttt{Boolean} \mid \ldots \\
\text{Value} & v ::= c^s \mid \lambda(\overline{x : \kappa}).\, e \mid \{\overline{f : v}\} \mid T\{\overline{f : v}\} \mid f \\
\text{Expression} & e ::= v \mid x \mid e(\overline{e}) \mid \oplus(\overline{e}) \mid \{\overline{f : e}\} \mid \theta_{p,t}(\overline{e}) \mid \\
& \qquad e.f \mid \texttt{table}(name) \mid \texttt{encr}(e, s) \mid \texttt{decr}(e) \\
\text{Prim. operator} & \oplus ::= + \mid - \mid \ldots \mid \wedge \mid \vee \mid \ldots \\
\text{Query operator} & \theta ::= \texttt{filter} \mid \texttt{proj} \mid \texttt{cross} \mid \texttt{agg} \mid \ldots
\end{array}
$$

Figure 5: Syntax and parameterization of Scylla's core query language. Terms/items in *red* backlit are security annotations not used by data analysts but generated automatically during query transformation. Similarly, table values in *blue* are only present at runtime. The superscript s in the base type d^s denotes either a plaintext ($s = \perp$) or encrypted ($s \in \mathcal{S}$) value of primitive type d. An overline represents a sequence.

confidentiality, under provider p. We use $e \Downarrow_{\mathsf{db}} v$ to denote that the result of executing Scylla query e, given inputs db, is v. Intuitively this property means that an adversary thwarted by any of mechanisms for level l, as per $\mathbb{P}$, observes indistinguishable values across two executions of a query (aka an expression e) on two stores, say db_1 and db_2, differing only in data with security level l. As input relations can contain many levels l, Def. 4.1 considers all levels l in $\mathcal{L}$ to ensure confidentiality guarantees for full query execution.

Our guarantee $\mathbb{P}$-NI differs from that of $\mathbb{S}$-NI, as the latter assumes mappings between levels and scheme-*domain* pairs, where domains consist of TEEs and providers. Our approach is more streamlined as it does not allow schemes and TEEs to be combined for the same data processing. We see very few cases where such layering substantially strengthens security guarantees, and believe that our more pragmatic choice simplifies the design of security policies. (Data encrypted with a given scheme can still transit through a TEE in Scylla without being decrypted if it is not being operated on in that TEE.)

We now provide proofs that our guarantees hold for all queries passing Scylla's type checking. The proof for Def. 4.1 makes use of the following result from the $\mathbb{S}$-NI formal framework [71, Th. 1], where all $\mathbb{S}$-NI entities conflicting in notation with $\mathbb{P}$-NI are annotated with the tilde ~ as shown in Thm. 4.1. We then formally connect the $\mathbb{P}$-NI and $\mathbb{S}$-NI frameworks using Lem. 4.2.

Theorem 4.1 ($\mathbb{S}$-NI Soundness). *If there exists non-function $\tilde{\kappa}$, s.t., $\rho \vdash_{\tilde{d}} \tilde{e} : \tilde{\kappa}$ w.r.t. $\mathbb{S}$ then $\mathbb{S}$-NI$(\tilde{e})_{\rho,\tilde{d}}$.*

Lemma 4.2 (Connecting $\mathbb{P}$-NI and $\mathbb{S}$-NI). *Expression e has $\mathbb{P}$-noninterference, i.e., $\mathbb{P}$-NI$(e)_{\rho,p}$ holds if $\mathbb{S}$-NI$(\tilde{e})_{\rho,\tilde{d}}$ holds, where $\mathfrak{T}(-)$ translates $\mathbb{P}$-NI entities to $\mathbb{S}$-NI entities.*

The set of encryption schemes $\mathcal{S}$ and set of security levels $\mathcal{L}$ are exactly the same for $\mathbb{P}$-NI as for $\mathbb{S}$-NI , but TEEs and providers are represented as $\mathbb{S}$-NI domains $\mathcal{D} = \mathcal{T}_\perp \times \mathcal{P}$. An overloaded $\mathfrak{T}(-)$ function translates $\mathbb{P}$-NI entities to $\mathbb{S}$-NI ones. The first step is the query translation: $\mathfrak{T}_p(e)$ takes a $\mathbb{P}$-NI query e (see Fig. 5) and the initial domain p producing a corresponding $\mathbb{S}$-NI expression

(see [71, Fig. 4]). The construction is defined via a "thunked" version $\mathfrak{T}'_p(e)$, i.e., $\mathfrak{T}_p(e) = \mathfrak{T}'_p(e)()$, which is mostly standard except for the following:

$$\begin{aligned}\mathfrak{T}'_p(\texttt{table}(name)) &= \lambda[(\rho^p(name), \bot)].\ \texttt{table}(name)\\ \mathfrak{T}'_p(\theta_{p',t}(\overline{e})) &= \lambda[(p', \bot)].\ (\lambda[(p', t)].\ \theta(\overline{\mathfrak{T}(e)}))()\end{aligned}$$

Using $\mathfrak{T}(e)$ we define the evaluation relation $e \Downarrow_{\mathsf{db}} v$ for any non-function v as $\mathfrak{T}(e) \rightarrow^*_{\mathsf{db}} v$. The validity of the definition comes from two facts: (1) the set of non-function values, including relations inside db, is the same for $\mathbb{P}$-NI and $\mathbb{S}$-NI ; (2) Thm. 4.1 only applies to non-function types. Due to the latter, we directly transfer $\mathsf{db}_1 \sim^l_\rho \mathsf{db}_2$ from $\mathbb{S}$-NI [71, Fig. 9] to $\mathbb{P}$-NI. For $v_1 \sim^{p,l} v_2$ we take the definition of the corresponding output relation from $\mathbb{S}$-NI [71, Fig.9], denoted here as $v_1 \approx^{d,l} v_2$ and replace the premise of EquivConst$^{\text{OUT}}$ with $(p, s) \in \mathbb{P}(l)$.

Finally, we describe the check that $\mathbb{P}$-NI uses on its queries that ultimately relies on $\mathfrak{T}(e)$ and on mapping $\mathbb{P}$-NI's security policy $\mathbb{P}$ into $\mathbb{S}$-NI 's policy $\mathbb{S} = \mathfrak{T}(\mathbb{P})$. Concretely, $\mathfrak{T}(\mathbb{P})$ performs the following transformation for each $l \in \mathcal{L}$: $\mathfrak{T}((d, \bot)) = ((\bot, p), \bot)$, $\mathfrak{T}((d, s)) = ((\bot, p), s)$ for $s \in \mathcal{S}$, and $\mathfrak{T}((d, t)) = ((t, p), s)$ for $t \in \mathcal{T}$. Now, we show that the Scylla check in $\mathbb{P}$-NI is $\mathbb{S}$-NI 's type check, i.e. we prove that:

Theorem 4.3. *If there exists non-function $\tilde{\kappa}$, s.t., $\rho \vdash_{(p,\bot)} \mathfrak{T}_p(e) : \tilde{\kappa}$ w.r.t. $\mathfrak{T}(\mathbb{P})$ then $\mathbb{P}$-NI$(e)_{\rho,p}$.*

Proof. The first step is applying Thm. 4.1 in order to infer $\mathbb{S}\text{-NI}(\mathfrak{T}_p(e))_{\rho,\tilde{d}}$. Then we use Lem. 4.2 to show $\mathbb{P}\text{-NI}(e)_{\rho,p}$ holds. By unrolling the definitions of $\mathbb{P}$-NI and $\mathbb{S}$-NI we infer that for respective results v_1, v_2 of $\mathbb{P}$-NI we have $v_1 \approx^{(p,\bot),l} v_2$. The latter can be seen to imply $v_1 \sim^{p,l} v_2$ by definition of $\mathfrak{T}(\mathbb{P})$. □

5 CONFIGURATION LANGUAGE

This section presents the API allowing Scylla to support a variety of mechanisms and security requirements at the query transformation and optimization stage (Spark SQL part of Fig. 4). Details about Spark SQL part, and aspects of execution like task scheduling and distribution (Spark core part of Fig. 4), are discussed in § 6.

5.1 Defining Mechanisms and Providers

List. 2 shows Scylla's API that defines the set of available providers and mechanisms. For a `Provider`, only its identity (its `name`) is needed for Spark's query transformation, the identity is then automatically linked with provider identity supplied to each `ScyllaWorker` and then exposed to SCM (see § 6) within Spark core. List. 2 includes `Client` and `Inet`, two built-in providers. Next, there are two kinds of `Mechanism`s: `TEE`s and encryption `Scheme`s. Scylla currently has two main interfaces for integrating TEEs, `VirtualTEE` and `NativeTEE` respectively, representing the two TEE flavors (see § 3): VM-like exemplified by `Nitro` and `SEV`, and enclave-like that are provided as a library, main example being `SGX`. `VirtualTEE`s' main characteristic is that they can run full-fledged Spark workers, and hence can benefit from the existing Spark infrastructure. Similar to providers, `VirtualTEE`'s identity is what primarily matters for the query execution pipeline. A `NativeTEE`, on the other hand, requires implementing query processing primitives using the TEE-specific API. To simplify the implementation effort of adding new `NativeTEE`s, Scylla includes TEE-independent support libraries covering most of the query execution logic. Finally, `Scheme` represents a *cryptographic system* (scheme) with the corresponding `encrypt` and `decrypt` methods. `Scheme`s may optionally support PHE operations by extending a respective `PHEOp` trait.

5.2 Heuristic-Based Mechanism Assignment

Based on the mechanisms and providers that are specified as per § 5.1 and confidentiality requirements captured by the security policy $\mathbb{P}$ (see § 4.2), Scylla transforms queries so that it exploits the specified mechanisms in the most efficient manner, while satisfying confidentiality constraints. The assignment of mechanisms and providers guides the whole process of query transformation. The said assignment is the task of an optimization *heuristic*.

Heuristic API. To capture the optimization variables the heuristic must find an assignment for (mechanisms and providers), Scylla extends vanilla Spark query representation (`LogicalPlan`) with annotations presented in List. 3. There, `Var[X]` is a generic trait for an optimization variable, which can either be `Val(v)`, i.e. set to some value `v` of type `X`, or `Free`, i.e., not yet set. `ScyllaExprAnnot` captures that each Scylla expression has a security label and is possibly encrypted with a scheme. Relational operators have neither their independent security label nor a single scheme, but are always executed in some provider and possibly using a TEE, both captured by `ScyllaPlanAnnot`. The annotations will be consumed by Scylla's further query transformation steps making direct use of the mechanism implementations (§ 5.1). The heuristic, represented by the `Heuristic` interface in List. 4, transforms a `ScyllaPlan` with all `Var`s being `Free` into a *fully annotated* `ScyllaPlan` with no `Free` choices left. Prior to execution, `ScyllaPlan` is always checked for security constraints using the type system as in Hydra [71], hence no heuristic can lead to violation of security guarantees.

The `Heuristic` interface captures the most general kind of mechanism assignment. It gives a lot of implementation freedom (e.g.,

```
trait Provider {val name: String}
object Inet extends Provider {... = "Inet"}
object Client extends Provider {... = "Client"}
sealed trait Mechanism {val name: String}
sealed trait TEE extends Mechanism
trait NativeTEE extends TEE
  def initialize(...)
  def setKeyInTEE(...)
  def changeEncryptionScheme(...)
  def project(...)
  ...
trait VirtualTEE extends TEE
trait Scheme extends Mechanism
  def encrypt(input: Long): Bytes
  def decrypt(input: Bytes): Long
sealed trait PHEOp
trait AddPHEOp extends PHEOp
  def add(x: Bytes, y: Bytes): Bytes
trait AddPtxtPHEOp extends PHEOp
  def addPtxt(x: Bytes, y: Long): Bytes
```

Listing 2: Core API for Scylla's execution capabilities.

even invoking an external optimization tool), but at the same requires substantial implementation effort and provides little help with ensuring that query semantics is preserved. Hence, Scylla also provides several simpler building blocks based on `ScyllaRule` and `ScyllaStep` interfaces inspired by Spark SQL's Catalyst. `ScyllaRule` is essentially `Heuristic` with a different contract: both input and output can have some but not necessarily all mechanisms assigned. `ScyllaStep` represents *local* and *conditional* (i.e., partial) modifications of a `ScyllaPlan`. `ScyllaRule` augments a traversal feature to a `ScyllaStep` (e.g., `.traverseUp` to traverse a `ScyllaPlan` tree). `RuleHeuristic` keeps applying the sequence of `ScyllaRules` to the `ScyllaPlan` until the fixed point is reached. In addition to normal traversals, `ScyllaPlan` provides equivalence-checking traversals with an `Eq` suffix (e.g., `.traverseUpEq`), which internally wraps every `ScyllaStep` into an `EqScyllaStep`. The latter additionally ensures that the transformation preserves semantics following the transformation relation $\rightsquigarrow$ introduced in $\mathbb{S}$-NI [71, Fig. 7].

Heuristic Examples. List. 5 shows heuristics from Hydra [71] and List. 6 shows a novel heuristic of Scylla; both are expressed using Scylla's API and DSL. The API uses Scala's standard library facility `andThen` to compose `ScyllaSteps` together. Individual `ScyllaSteps` can be trivially lifted from predicates and total functions via `check(..)` and `rule(..)`, respectively. To select minimum cost `ScyllaStep`, we use `minCost(..)`. The cost is computed from execution times of encryption schemes and operators in microbenchmarks (see § 2). Finally, Scylla provides `setSchemesMinCost()`, which assigns schemes used by expressions within a given query operator to minimize the cost, and if no assignment satisfies $\mathbb{P}$ while only using operations supported by `Schemes`, the `ScyllaStep` is undefined.

```
trait ScyllaExprAnnot
  val label: Label
  val scheme: Var[Option[Scheme]]
trait ScyllaPlanAnnot
  val provider: Var[Provider]
  val tee: Var[Option[TEE]]
class ScyllaPlan extends LogicalPlan
    with ScyllaPlanAnnot {...}
class ScyllaExpression extends Expression
    with ScyllaExprAnnot {...}
```

Listing 3: Scylla's annotations for Spark's logical plan and expressions thereof. `Vars` are to be filled in by the heuristic.

```
trait Heuristic
  // p ≈ LogicalPlan: has only free vars
  def apply(p: ScyllaPlan): ScyllaPlan
type ScyllaStep =
  PartialFunction[ScyllaPlan, ScyllaPlan]
type ScyllaRule = ScyllaPlan => ScyllaPlan
class RuleHeuristic(rs: ScyllaRule*)
  extends Heuristic {...}
trait EqScyllaStep extends ScyllaStep
  def safeApply(p: ScyllaPlan): ScyllaPlan
  override def isDefinedAt(p: ...): Boolean
  final def apply(p: ScyllaPlan) =
    {/*checks p⇝safeApply(p) [71, Fig.7]*/}
```

Listing 4: Scylla's heuristic API and auxiliary structures.

6 SCYLLA RUNTIME DESIGN

Scylla's runtime introduces secure components that span over Spark SQL's query execution pipeline, Spark core and the Standalone cluster manager (see Fig. 2).

6.1 Scylla Catalyst Integration

Fig. 6 presents details of Scylla's query transformation integrated into Catalyst query optimizer. The transformation ranges over custom logical plan optimization rules applied to a Scylla logical plan, specialized strategies for Scylla query planner to generate a mechanism-aware Scylla physical plan, and custom query preparation rules applied to the physical plan to generate the final *executed* physical plan. Scylla leverages security-related metadata and operations to augment existing entities of the query execution pipeline to get to an annotated logical plan `ScyllaPlan` introduced in § 5.2, and a similarly annotated physical plan `ScyllaPlanExec`.

```
cldR: ScyllaRule = _.transformUpEq(
  check(_.provider.isFree
    && _.children.all(_.provider == AWS)
  ) andThen rule(_.setP(AWS).setT(None))
    andThen setSchemesMinCost()
)
restSGXR: ScyllaRule = _.transformUpEq(
  check(_.provider.isFree)
    andThen rule(_.setP(AWS).setT(SGX))
    andThen setSchemesMinCost()
)
restClientR: ScyllaRule = _.transformUpEq(
  check(_.provider.isFree)
    andThen rule(_.setP(Client).setT(None))
    andThen setSchemesMinCost()
)
hydraSGX = RuleHeuristic(restSGXR)
hydraPHE = RuleHeuristic(cldR, restClientR)
hydraHybrid = RuleHeuristic(cldR, restSGXR)
```

Listing 5: Heuristics from Hydra [71] expressed conveniently with `ScyllaRules`.

```
cldWithSGXR: ScyllaRule = _.transformUpEq(
  check(!_.provider.isFree)
  andThen minCostOf(
    rule(_.setP(Azure).setT(None))
      andThen setSchemesMinCost(),
    rule(_.setP(Azure).setT(SGX))
      andThen setSchemesMinCost()
  )
)
cldWithSGX = RuleHeuristic(cldWithSGXR)
```

Listing 6: Rule-based heuristic: compute in the cloud (`Azure`) either directly via PHE or via `SGX`, *in any order* (cf. `hydraHybrid` from List. 5).

In Fig. 6, a Heuristic chooses the *mechanisms* while *optimizing* the expected performance of query execution. This results in annotations on ScyllaPlan that captures everything affecting security properties. The next step introduces *PHE expressions*. Standard expressions for operations, such as Add and Multiply, having scheme-annotated arguments are replaced with Scylla's counterparts PHEAdd and PHEMultiply. The latter delegate computation to AddPHEOp and MultiplyPHEOp methods of the respective Schemes. Scylla then *validates* the annotations using our type checking algorithm based on Hydra [71] to ensure that there are no violations as per security policy $\mathbb{P}$ and establish $\mathbb{P}$-NI. Type checking also validates that PHE operations requested by the query's annotations are implemented by the corresponding Schemes, e.g., if there is an addition of two encrypted expressions, the corresponding Scheme class implements PHEAdd. Only on successful validation, denoted by green color in Fig. 6, transformation proceeds further.

Remaining query transformation steps inspect the security annotations and modify the query to actually use the mechanisms. These steps only rely on the interfaces outlined in § 5.1 and do not need be changed when extending Scylla with new mechanisms or heuristics. Then, Scylla query planner transforms the optimized secure logical plan ScyllaPlan (ScP) into a physical plan ScyllaPlanExec (ScPP). In the process, annotated relational operators are replaced with Scylla's *generic relational operators*, which either target VM-like or enclave-like TEEs. In all cases, the introduced operators keep Provider metadata, but they execute differently depending on whether tee refers to NativeTEE or not (i.e., absent or VirtualTEE). For a NativeTEE, the operator's execute() method invokes an appropriate function of NativeTEE trait (e.g., relational projection would call project). Otherwise, the operator reuses the existing Spark's execute() and, if VirtualTEE annotation is present, augments provider metadata with the identity of the TEE. Both pieces of metadata are propagated to security mechanism-aware RDDs, which later signal the scheduler in Spark core to place the corresponding computation onto an appropriate executor.

The remaining two steps prepare the ScyllaPlanExec (ScPP) for execution. First, secure *exchange operators* are introduced to ensure that the stages of query execution belonging to different providers and VirtualTEEs are represented as separate stages of computation and, hence, can be scheduled at appropriate executors. Second, *encryption/decryption operators* are inserted at appropriate places to ensure that the data crossing provider boundaries is encrypted according to the Inet provider and that the data entering (resp. leaving) VirtualTEEs is decrypted (resp. encrypted).

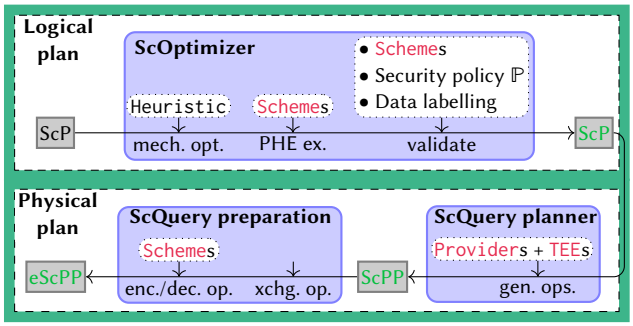

Figure 6: Steps of query transformation in Spark SQL.

6.2 Scylla Spark Core Integration

The *executed* physical plan (eScPP) is then handed over to the customized Spark core for further processing. Tab. 1 shows components of Spark core and standalone cluster manager that are modified, extended, or replaced with custom mechanism-aware implementations. Scylla's efficient design introduces minimal changes to the existing Spark core. Spark context is switched out with a novel Scylla context (SC) having custom components – Scylla DAG scheduler (SDS), Scylla task scheduler (STS), Scylla scheduler backend (SSB), which respectively replace existing ones – DAG scheduler, task scheduler, scheduler backend.

Query Execution. Scylla session (SS) is the entry point for programming with the DataFrame API – the main abstraction of Spark SQL for executing relational workloads. The SC object associated with a SS is the heart of a Spark application. The driver creates and uses SC to coordinate the running of the Spark application on the cluster. SC communicates with the Scylla cluster manager (SCM), prompting the latter to allocate executors to the application on the nodes across the cluster. SCM is based on Spark's standalone cluster manager. Once executors are launched, application code is shipped to them as defined by the JAR passed to SC. Finally, SC sends tasks to the executors who run the tasks and return the results.

SDS is the high-level scheduling layer that implements stage-oriented scheduling. It computes a DAG of mechanism-aware stages for each submitted job, keeps track of which RDDs and stage outputs are materialized, and finds a minimal schedule to run the job. It then submits stages as mechanism-aware task sets to an underlying STS implementation that runs them securely on the cluster. Mechanism information is injected from the stage into the task sets as they are being created. STS schedules tasks for the cluster by acting through a scheduler backend which is implemented concretely as Scylla coarsegrained scheduler backend (SCGSB). STS receives sets of mechanism aware tasks submitted to them from SDS for each stage, and is responsible for sending the tasks to the cluster, running them, retrying if there are failures, and mitigating stragglers. STS delegates the responsibility of launching tasks on executors possessing the correct mechanism to the Scylla taskset manager (STSM).

Task Scheduling, Distribution, Launching, Execution. SCGSB acts as a bridge between the driver and the worker nodes. It receives

Table 1: Lines of code per Scylla component. Insertions(+), deletions(-), modifications(!) are w.r.t. original Spark components.

Scylla component	LoC	(+)	(-)	(!)
Cluster	724	173	68	161
Driver	2782	162	1048	302
Deployment	2110	110	269	175
Executor	218	11	50	74
Spark SQL	43	0	1074	13
Spark	256	11	50	74

resource offers from the worker nodes and coordinates task scheduling. SCGSB interacts with the SCM to request resources and allocate tasks. It utilizes the resource management capabilities provided by the cluster manager to schedule tasks across the cluster. SCGSB initiates the the process of generating resource offers based on the available resources on worker nodes. It communicates the resource offers from workers to the STS and receives the active mechanism-aware task sets for tasks in order of priority. It then allocates mechanism-aware tasks to worker nodes with appropriate mechanism based on priority and resource availability. Internally, STS responds with the active mechanism-aware task sets by virtue of the STSM, which, under the hood, ensures that a correct mechanism-aware task is appropriately issued in response to an offer of a single executor possessing a mechanism from the SCGSB. This sort of directed scheduling w.r.t. mechanisms ensures that tasks are scheduled by STS strictly as per security constraints and only then SCGSB communicates the task descriptions to Scylla coarsegrained executor backend (SCGEB) running on each executor. SCGEB receives the task descriptions and launches the tasks on the executor. The invariant that all tasks (narrow transformations/operators) in one stage have the same mechanism is ensured. At runtime, a special mechanism change operator (invisible to the data analyst) introduces a new stage when a new mechanism (as decided by the heuristic) is to be assigned to an operator.

Nitro Remote Communication. NEs are highly isolated, with no durable storage, no network access, no interactive access, no metadata serivces/DNS/NTP. All communication between the NE and the instance is via a bi-directional VM socket (vsock) which supports local streaming (TCP-like) communication that can not be leaked, spoofed, or intercepted. Hence, the only way to communicate with a NE is using a vsock socket. This special type of communication mechanism acts as an isolated communication channel between the parent EC2 instance and NEs. Since a NE is not connected to the outside world, we use TSI to enable communication channels among Scylla application runtime components (custom Scylla workers and Scylla executors) hosted in all NEs running on the cluster. All communication is channeled through vsock using socat [67] bridges between the entities within the pairs (parent host, vsock device of NE) and (vsock device of parent VM, NE) as shown in the left (white) and right (red) portions respectively of Fig. 7.

7 EVALUATION

With Scylla's security guarantees (S) defined in § 4.2, and users writing queries agnostically to security constraints (T), we evaluate different facets of Scylla's T and E empirically by addressing the following research questions:

- **RQ1**: How does Scylla perform w.r.t. to state-of-the-art systems?
- **RQ2**: How independent is Scylla w.r.t. different mechanisms and how do these fare w.r.t. each other?
- **RQ3**: How substantial are the performance benefits of novel heuristics?

Benchmarks. We focus on the industry-standard TPC-H benchmark [102], due to its wide application and use in systems we compare against, and in many others (e.g., [88, 92, 104]). All reported end-to-end execution times are averages of 5 runs, and all overheads are computed using geometric mean. Execution times are clocked from the point of query submission on encrypted data until the point results are decrypted on the driver/client-side, and hence do not include one-time costs for data preparation, infrastructure setup, RA of enclaves, and provisioning cryptographic keys. Since RA's latency is low (~hundreds of milliseconds), we do not consider it in our query execution times.

Setup. We use two state-of-the-art systems, Opaque [120] and Hydra [71], both Spark-based like Scylla, for comparative evaluation. Opaque uses SGX for confidentiality and is at the center of a commercial product [82]. *Oblivious RAM* (ORAM) [46], which is optionally supported by Opaque with a set of oblivious operations for preventing information leakage through access patterns, was disabled for fair comparison. Hydra supports both SGX and PHE and a limited combination of the two. Hydra has been shown to be faster than PHE-only system Cuttlefish [93] when using the same PHE schemes, and faster than both Cuttlefish and Opaque when combining SGX and PHE. TPC-H [102] was used (as the only benchmark) to evaluate both Opaque SQL [105] and Hydra (including w.r.t. Opaque).

We run experiments on TPC-H (scale factor 10 for plaintext) using software mechanisms (PHE) and hardware mechanisms (SGX, SEV, TDX, Nitro), in two cloud environments as none provides *all* the hardware mechanisms supported by Scylla. Our cluster comprises of 5 VMs for the untrusted cloud and 1 VM for the trusted driver/client-side. All VMs were running Ubuntu 22.04.3 LTS with Linux 6.5. From Amazon AWS, we use r5.4xlarge VMs for all workloads. The VMs were appropriately enabled with AWS Nitro (Nitro CLI v1.3, Docker v23.0) as required. From Azure we use DC16ads_v5 VMs which use AMD's 3^{rd}-generation EPYC processor to offer SEV-SNP, DC16es_v5 VMs powered by 4^{th} Intel Xeon scalable processors providing Intel TDX, and DC16ds_v3 VMs powered by the latest 3^{rd} generation Intel Xeon scalable processors running SGX SDK v2.21.

Security policy used (along the lines of List. 1) for evaluation ensured that plaintext data was encrypted with same schemes as done in Hydra and Opaque to keep comparisons fair in research

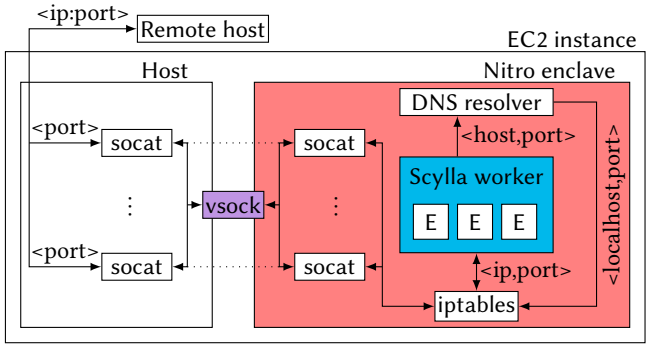

Figure 7: Communication between Nitro enclave and remote host is channeled through vsock using socat bridges on the host and in the enclave. Sockets in the host and nitro enclave are transparently bridged to AF_VSOCK socket (vsock) - the special Linux socket for VM-host communication.

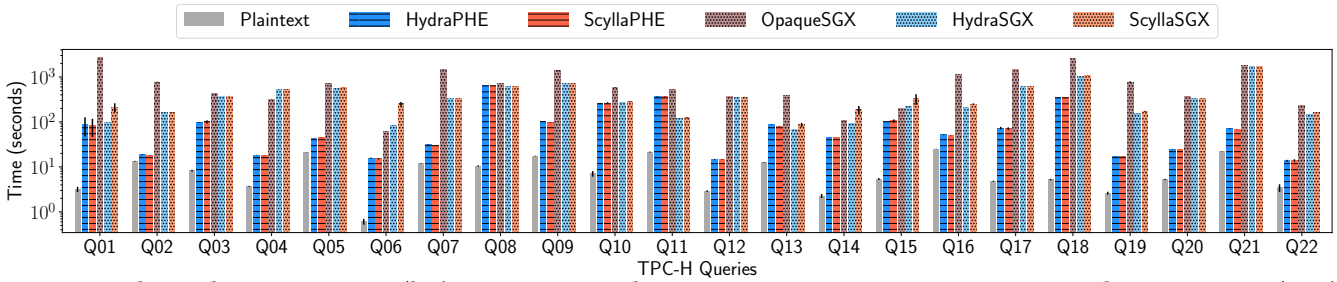

Figure 8: End to end execution times (log). $\text{Scylla}_{\text{PHE}}$, $\text{Hydra}_{\text{PHE}}$ in AWS; $\text{Opaque}_{\text{SGX}}$, $\text{Scylla}_{\text{SGX}}$, $\text{Hydra}_{\text{SGX}}$ in Azure (RQ1).

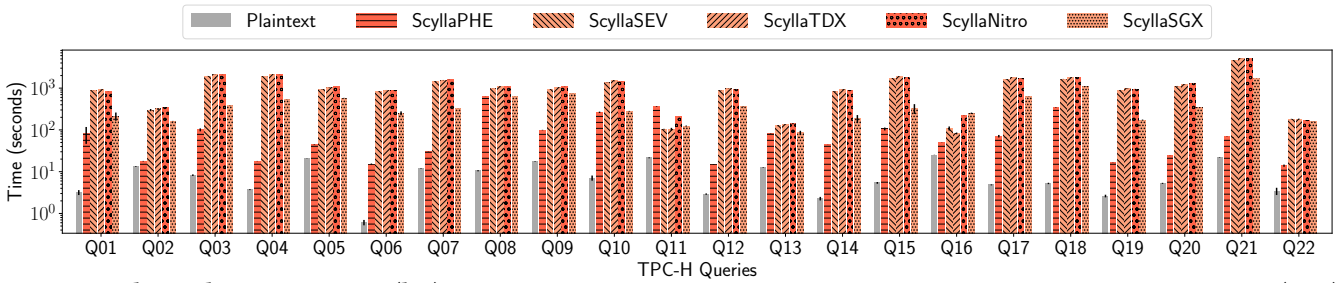

Figure 9: End to end execution times (log). $\text{Scylla}_{\text{Nitro}}$, $\text{Scylla}_{\text{PHE}}$ in AWS; $\text{Scylla}_{\text{SGX}}$, $\text{Scylla}_{\text{SEV}}$, $\text{Scylla}_{\text{TDX}}$ in Azure (RQ2).

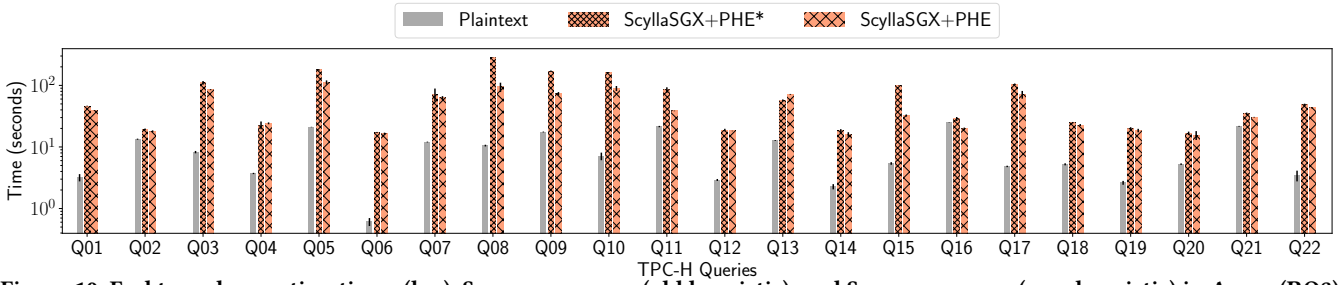

Figure 10: End to end execution times (log). $\text{Scylla}_{\text{SGX+PHE*}}$ (old heuristic), and $\text{Scylla}_{\text{SGX+PHE}}$ (new heuristic) in Azure (RQ3).

questions across all relevant *(execution) modes*. Data for modes SGX+PHE and PHE was encrypted with same schemes respectively as done in Hydra. Data for modes SGX, SEV, TDX, and Nitro, was encrypted with AES-GCM [73] following what was done in Hydra and Opaque for SGX mode. Our hybrid heuristics (in short, when to favor PHE over TEEs) are guided by execution times overheads of running SQL operators as per Fig. 1.

RQ1: Scylla, Opaque, and Hydra with Single Mechanisms (PHE, SGX) on TPC-H. Fig. 8 compares Hydra, Scylla, and Opaque using single mechanisms as supported respectively (with corresponding heuristics for Hydra and Scylla). On average, $\text{Scylla}_{\text{PHE}}$ is 1.02× faster than $\text{Hydra}_{\text{PHE}}$, and $\text{Scylla}_{\text{SGX}}$ is 1.91× faster than $\text{Opaque}_{\text{SGX}}$ and $\text{Hydra}_{\text{SGX}}$ is 1.19× faster than $\text{Scylla}_{\text{SGX}}$. Scylla like Hydra uses custom serialization and different SGX libraries w.r.t. Opaque (Intel SGX SDK as opposed to Open Enclave SDK). Importantly, Scylla is at least as fast as Hydra for PHE mode and faster than Opaque for SGX mode, confirming that its general approach (I) is competitive adding a minor overhead only occasionally (e.g., w.r.t. $\text{Hydra}_{\text{SGX}}$).

RQ2: Scylla with Single Mechanisms (TEEs SGX, SEV, TDX, Nitro; and PHE) on TPC-H. Fig. 9 compares the different TEEs, as well as PHE, in Scylla. On average, $\text{Scylla}_{\text{PHE}}$ is 5.61× faster than $\text{Scylla}_{\text{SGX}}$, which is 2.37× faster than $\text{Scylla}_{\text{SEV}}$, which in turn is 1.15× and 1.07× faster than $\text{Scylla}_{\text{Nitro}}$ and $\text{Scylla}_{\text{TDX}}$ respectively. $\text{Scylla}_{\text{TDX}}$ is 1.08× faster than $\text{Scylla}_{\text{Nitro}}$. Modes SEV, TDX, and Nitro are close in performance but slower than SGX mode due to encryption/decryption of data exiting/entering the enclaves happening in JVM in the former three compared to native code in SGX mode. Nitro's remote communication overhead also makes it slightly slower than SEV. The superior performance of PHE mode is mostly due to direct computation on ciphertext without having to decrypt → compute → encrypt the data as done in TEEs (cf. § 2).

RQ3: Scylla and Hydra with Multi-Mechanism Heuristics. The novel heuristic dubbed SGX+PHE (aka `cldWithSGX` from List. 6 which selects the best out of PHE or SGX based on latencies in Fig. 1) is compared with a simpler heuristic dubbed SGX+PHE* used in Hydra (aka `hydraHybrid` from List. 5 which by default naively selects PHE until forced to switch to SGX). Data encrypted with same schemes was used for both modes SGX+PHE and SGX+PHE* to ensure fairness. As shown in Fig. 10, on average, $\text{Scylla}_{\text{SGX+PHE}}$ is 1.36× faster than $\text{Scylla}_{\text{SGX+PHE*}}$. This shows how small changes in heuristics to choose the mechanism with lower execution latency for each operator can have an impact on performance. Scylla supports such changes easily by its ability to quickly devise new heuristics like SGX+PHE using its APIs and DSL.

8 RELATED WORK

Scylla builds on previous works connected to container security [8, 17, 18, 94, 119], flexible TEE architectures [14], access control models for map-reduce based systems [90, 107], Hadoop ecosystem [52, 53], big data [33, 35] and big data platforms such as Spark [106, 113, 114], and databases [24, 34]. Scylla currently focuses on automating the use of TEEs (enclave-like `NativeTEEs` and CVMs/`VirtualTEEs`), in the context of confidential computing with a single-party addressing many existing use cases as opposed to approaches like secure multi-party computing [26, 41, 121] which address challenges in multi-party scenarios.

Secure Data Analytics and Data Processing. Several works focus on securing databases, using different single mechanisms and underlying database systems. Examples include CryptDB [86] (PHE, MySQL), TrustedDB [15] (IBM4764 secure co-processor, MySQL), Cipherbase [7] (FPGA-based trusted component, Microsoft SQL Server), and Monomi [104] (PHE, Postgres). None of these allow mechanisms to be easily changed, and none support scalable distributed processing required for big data analytics. Autocrypt [101] similarly uses PHE for securing web servers.

Several works extend Spark with individual/fixed mechanisms. Before Opaque extended Spark with SGX and ORAM, Seabed [85] extended Spark with PHE including novel symmetric schemes. Symmetria [92] extended Spark with yet more efficient novel symmetric PHE schemes. Cuttlefish [93] extended Spark with both PHE and SGX, leveraging the latter only for re-encryption though. Like Opaque, Flare [64] extends Spark for use with SGX, reducing TCB and adding SGX-specific optimizations.

Scylla has a number of advantages over Hydra[71]. Scylla presents a novel system along with a runtime architecture that supports both VM-like (`VirtualTEE`) and enclave-like (`NativeTEE`) TEEs, also being the first system (to the best of our knowledge) to leverage Nitro at scale for confidential computing. Scylla provides a more streamlined simplified security policy and model, w.r.t. Hydra, which also allows Scylla to reason about initial and interim (inter-node communication) encryption schemes for data, while benefiting from the same rigorous guarantees. Scylla also presents a novel API design e.g. including TEEs at different "levels", and demonstrates easy portability to CVMs, along with a DSL, and flexible rich heuristics. Hydra focussed on the formal framework – language, type system, evaluation semantics, query transformation, and establishing a noninterference guarantee. Scylla allows mechanisms to be assigned more flexibly thus allowing more elaborate heuristics leading to improved performance as demonstrated.

Access Control for Big Data. Many works provide access control for big data processing, without strong (cryptographic) enforcement, using software or hardware mechanisms to protect data in use. Vigiles [107] enforces access control for all types of data (e.g., structured, unstructured, semi-structured), in the map-reduce model without requiring any modification to the source code of map-reduce systems. Vigiles automatically rewrites the cloud's front-end API by augmenting them with reference monitors. Airavat [90] integrates mandatory access control and *differential privacy* (DP) [37] in map-reduce computations. GuardMR [106] adds access control at the level of key-value pairs for map-reduce. HeABAC [52] provides attribute-based access control for multi-tenant Hadoop deployments. SparkXS [87] provides access control for streaming data in Spark. SparkAC [113] augments Spark with *purpose-aware access control* (PAAC) [32] using GuardSpark++ [114] in Spark Catalyst. *Access control model for Hadoop ecosystem* (HeAC) [53] is a formal authorization model proposed for Apache Sentry and Ranger to authorize object accesses based on object attributes (tags) for various systems with diverse objects. Also formal *attribute-based access control* (ABAC)-based techniques like [52] have been proposed for context-based access control in Hadoop ecosystem projects.

Mechanism-Independent Systems. Several works propose generic extensions of TEEs. Like works strengthening individual TEEs, these are complementary to Scylla. Enarx [39] is a CPU-architecture and cloud provider neutral framework to simplify application deployment transparently to a variety of TEEs in the cloud by using *WebAssembly* (WASM) and a microkernel. Enarx is suitable for lightweight workloads and would incur a high overhead for running complex general-purpose confidential computing workloads with Spark. Veracruz [110] is designed for multi-party collaborative computation, and its reliance on WASM, similar to Enarx, likely makes it more suitable for lightweight workloads. Unlike Scylla, Enarx supports SGX and SEV; Veracruz supports SGX, SEV, and TrustZone. Nimble [5] proposes rollback protection for confidential cloud services which is TEE-independent but focuses on a very specific guarantee – adding protection against rollback attacks [72] where the adversary violates the integrity of a protected application state by replaying old persistently stored data or by starting multiple application instances. Cerberus [61], a formal approach for secure and efficient enclave memory sharing, shows that memory sharing can substantially improve performance, and provides formal guarantees about security – establishing the *secure remote execution* (SRE) property via automated formal verification. Cerberus proposes a general formal enclave platform model with memory sharing that weakens the disjoint memory assumption and captures a family of enclave platforms. *PoBF-compliant framework* (PoCF) [28] is a framework for *confidential computing as a service* (CCaaS), based on the security objective of *proof of being forgotten* (PoBF), to verify the behavior of in-TEE programs in order to prevent them from unintentionally leaking sensitive data and residue threats. PoCF consists of a verifier based on Rust's type system and security features to prove PoBF-compliance for the whole enclave program. While PoCF is designed with a TEE-agnostic library, its CCaaS prototypes/PoCF enclaves are currently implemented only for SGX and SEV.

9 CONCLUSIONS

We proposed a mechanism-independent confidential analytics framework Scylla with a novel general architecture, implemented by extending Spark, that can transparently utilize an extensible set of software mechanisms and hardware mechanisms (both VM-like and enclave-like). Scylla's formal underpinnings ensure a novel security property based on noninterference. Avenues for future research include considering stronger adversaries, extending our approach into streaming applications, distinguishing between different parties, and incorporating security notions like differential privacy.

REFERENCES

[1] [n. d.]. Fortanix. https://www.fortanix.com.

[2] Adil Ahmad, Alex Schultz, Byoungyoung Lee, and Pedro Fonseca. 2023. An Extensible Orchestration and Protection Framework for Confidential Cloud Computing. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, Roxana Geambasu and Ed Nightingale (Eds.). USENIX Association, 173–191. https://www.usenix.org/conference/osdi23/presentation/ahmad

[3] AMD. 2020. AMD SEV-SNP. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf.

[4] AMD. 2021. AMD Secure Encrypted Virtualization. https://www.amd.com/en/developer/sev.html.

[5] Sebastian Angel, Aditya Basu, Weidong Cui, Trent Jaeger, Stella Lau, Srinath T. V. Setty, and Sudheesh Singanamalla. 2023. Nimble: Rollback Protection for Confidential Cloud Services. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, Roxana Geambasu and Ed Nightingale (Eds.). 193–208. https://www.usenix.org/conference/osdi23/presentation/angel

[6] Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. 2015. S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing - and Its Application to AES. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. 591–604. https://doi.org/10.1109/SP.2015.42

[7] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. 2013. Orthogonal Security with Cipherbase. http://www.cidrdb.org/cidr2013/Papers/CIDR13_Paper33.pdf

[8] 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 *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 689–703. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov

[9] AWS. 2019. Security benefits of the Nitro architecture. https://www.youtube.com/watch?v=0qcUOKupt7Y.

[10] AWS. 2020. Deep dive on AWS Nitro Enclaves for applications running on Amazon EC2. https://www.youtube.com/watch?v=yDe_C_fpkfg.

[11] AWS. 2022. C5 Instances and the Evolution of Amazon EC2 Virtualization. https://www.youtube.com/watch?v=LabltEXk0VQ.

[12] AWS. 2022. Powering Amazon EC2: Deep dive on the AWS Nitro System. https://www.youtube.com/watch?v=jAaqfeyvvSE.

[13] AWS. 2024. The Security Design of the AWS Nitro System. https://docs.aws.amazon.com/pdfs/whitepapers/latest/security-design-of-aws-nitro-system/security-design-of-aws-nitro-system.pdf.

[14] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf. 2021. CURE: A Security Architecture with CUstomizable and Resilient Enclaves. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael D. Bailey and Rachel Greenstadt (Eds.). 1073–1090. https://www.usenix.org/conference/usenixsecurity21/presentation/bahmani

[15] Sumeet Bajaj and Radu Sion. 2011. TrustedDB: A Trusted Hardware Based Database with Privacy and Data Confidentiality. 205–216. http://doi.acm.org/10.1145/1989323.1989346

[16] Jeff Barr. 2018. AWS News Blog. Amazon EC2 Update – Additional Instance Types, Nitro System, and CPU Options. https://aws.amazon.com/blogs/aws/amazon-ec2-update-additional-instance-types-nitro-system-and-cpu-options/.

[17] Andrew Baumann, Marcus Peinado, and Galen C. Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven. 267–283. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/baumann

[18] Ferdinand Brasser, Patrick Jauernig, Frederik Pustelnik, Ahmad-Reza Sadeghi, and Emmanuel Stapf. 2022. Trusted Container Extensions for Container-based Confidential Computing. *CoRR* abs/2205.05747 (2022). https://doi.org/10.48550/arXiv.2205.05747 arXiv:2205.05747

[19] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). 991–1008. https://www.usenix.org/conference/usenixsecurity18/presentation/bulck

[20] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. 54–72. https://doi.org/10.1109/SP40000.2020.00089

[21] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution, SysTEX@SOSP 2017, Shanghai, China, October 28, 2017*. 4:1–4:6. https://doi.org/10.1145/3152701.3152706

[22] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2018. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). 178–195. https://doi.org/10.1145/3243734.3243822

[23] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. 1041–1056. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/van-bulck

[24] Ji-Won Byun and Ninghui Li. 2008. Purpose based access control for privacy protection in relational database systems. *VLDB J.* (2008), 603–619. https://doi.org/10.1007/s00778-006-0023-0

[25] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. 2019. Fallout: Leaking Data on Meltdown-resistant CPUs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). 769–784. https://doi.org/10.1145/3319535.3363219

[26] Edward Chen, Jinhao Zhu, Alex Ozdemir, Riad S. Wahby, Fraser Brown, and Wenting Zheng. 2023. Silph: A Framework for Scalable and Accurate Generation of Hybrid MPC Protocols. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 848–863. https://doi.org/10.1109/SP46215.2023.10179397

[27] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten-Hwang Lai. 2020. SgxPectre: Stealing Intel Secrets From SGX Enclaves via Speculative Execution. *IEEE Secur. Priv.* (2020), 28–37. https://doi.org/10.1109/MSEC.2019.2963021

[28] Hongbo Chen, Haobin Hiroki Chen, Mingshen Sun, Kang Li, Zhaofeng Chen, and XiaoFeng Wang. 2023. A Verified Confidential Computing as a Service Framework for Privacy Preservation. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, Joseph A. Calandrino and Carmela Troncoso (Eds.). 4733–4750. https://www.usenix.org/conference/usenixsecurity23/presentation/chen-hongbo

[29] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David F. Oswald, and Flavio D. Garcia. 2021. VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael D. Bailey and Rachel Greenstadt (Eds.). 699–716. https://www.usenix.org/conference/usenixsecurity21/presentation/chen-zitai

[30] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. 2020. TFHE: Fast Fully Homomorphic Encryption Over the Torus. *Journal of Cryptology* 33 (2020), 34–91.

[31] Pavel Chuprikov, Patrick Eugster, and Shamiek Mangipudi. 2025. Security Policy as Code. *IEEE Secur. Priv.* 23, 2 (2025), 23–31. https://doi.org/10.1109/MSEC.2025.3535803

[32] Pietro Colombo and Elena Ferrari. 2015. Privacy Aware Access Control for Big Data. *Big Data Res.* 2, 4 (dec 2015), 145–154.

[33] Pietro Colombo and Elena Ferrari. 2015. Privacy Aware Access Control for Big Data: A Research Roadmap. *Big Data Res.* (2015), 145–154. https://doi.org/10.1016/j.bdr.2015.08.001

[34] Pietro Colombo and Elena Ferrari. 2017. Enhancing MongoDB with Purpose-Based Access Control. *IEEE Trans. Dependable Secur. Comput.* (2017), 591–604. https://doi.org/10.1109/TDSC.2015.2497680

[35] Pietro Colombo and Elena Ferrari. 2018. Access Control in the Era of Big Data: State of the Art and Research Directions. In *Proceedings of the 23nd ACM on Symposium on Access Control Models and Technologies, SACMAT 2018, Indianapolis, IN, USA, June 13-15, 2018*, Elisa Bertino, Dan Lin, and Jorge Lobo (Eds.). 185–192. https://doi.org/10.1145/3205977.3205998

[36] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. 2018. CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* (2018), 171–191. https://doi.org/10.13154/TCHES.V2018.I2.171-191

[37] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam D. Smith. 2016. Calibrating Noise to Sensitivity in Private Data Analysis. *J. Priv. Confidentiality* 7, 3 (2016), 17–51. https://doi.org/10.29012/JPC.V7I3.405

[38] T. ElGamal. 1985. A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. *Trans. on Information Theory* 31, 4 (1985), 469–472.

[39] Enarx Project. Confidential Computing Consortium. Linux Foundation. 2023. Enarx: Confidential Computing with WebAssembly. https://enarx.dev/.

[40] M. Falkner and J. Apostolopoulos. 2022. Intent-based Networking for the Enterprise: A Modern Network Architecture. *Commun. ACM* 65, 11 (2022), 108–117.

[41] Vivian Fang, Lloyd Brown, William Lin, Wenting Zheng, Aurojit Panda, and Raluca Ada Popa. 2022. CostCO: An automatic cost modeling framework for secure multi-party computation. In *7th IEEE European Symposium on Security and Privacy, EuroS&P 2022, Genoa, Italy, June 6-10, 2022*. IEEE, 140–153. https://doi.org/10.1109/EUROSP53844.2022.00017

[42] Craig Gentry. 2009. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing (STOC '09)*. ACM, New York, NY, USA, 169–178. https://doi.org/10.1145/1536414.1536440

[43] Craig Gentry and Shai Halevi. 2011. Implementing Gentry's Fully-homomorphic Encryption Scheme. In *Proceedings of the 30th Annual International Conference on Theory and Applications of Cryptographic Techniques: Advances in Cryptology (EUROCRYPT'11)*. Springer-Verlag, Berlin, Heidelberg, 129–148. http://dl.acm.org/citation.cfm?id=2008684.2008697

[44] Craig Gentry, Shai Halevi, and Nigel P. Smart. 2012. Homomorphic Evaluation of the AES Circuit. *IACR Cryptol. ePrint Arch.* (2012), 99. http://eprint.iacr.org/2012/099

[45] Joseph A Goguen and José Meseguer. 1982. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*. IEEE, 11–20.

[46] Oded Goldreich. 1987. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In *ACM Symposium on Theory of Computing (STOC '87)*. 182–194. https://doi.org/10.1145/28395.28416

[47] Christian Göttel, Rafael Pires, Isabelly Rocha, Sébastien Vaucher, Pascal Felber, Marcelo Pasin, and Valerio Schiavoni. 2018. Security, Performance and Energy Trade-Offs of Hardware-Assisted Memory Protection Mechanisms. In *37th IEEE Symposium on Reliable Distributed Systems, SRDS 2018, Salvador, Brazil, October 2-5, 2018*. IEEE Computer Society, 133–142. https://doi.org/10.1109/SRDS.2018.00024

[48] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). 955–972.

[49] Brendan Gregg. 2017. AWS EC2 Virtualization 2017: Introducing Nitro. https://www.brendangregg.com/blog/2017-11-29/aws-ec2-virtualization-2017.html.

[50] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings (Lecture Notes in Computer Science)*, Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez (Eds.). 279–299. https://doi.org/10.1007/978-3-319-40667-1_14

[51] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, Jaeyeon Jung and Thorsten Holz (Eds.). 897–912. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss

[52] Maanak Gupta, Farhan Patwa, and Ravi Sandhu. 2018. An Attribute-Based Access Control Model for Secure Big Data Processing in Hadoop Ecosystem. In *Proceedings of the Third ACM Workshop on Attribute-Based Access Control (ABAC'18)*. New York, NY, USA, 13–24. https://doi.org/10.1145/3180457.3180463

[53] Maanak Gupta, Farhan Patwa, and Ravi S. Sandhu. 2017. Object-Tagged RBAC Model for the Hadoop Ecosystem. In *Data and Applications Security and Privacy XXXI - 31st Annual IFIP WG 11.3 Conference, DBSec 2017, Philadelphia, PA, USA, July 19-21, 2017, Proceedings (Lecture Notes in Computer Science)*, Giovanni Livraga and Sencun Zhu (Eds.). 63–81. https://doi.org/10.1007/978-3-319-61176-1_4

[54] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. 2020. Bluethunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* (2020), 321–347. https://doi.org/10.13154/TCHES.V2020.I1.321-347

[55] IBM. 2024. Cost of a Data Breach Report. https://wp.table.media/wp-content/uploads/2024/07/30132828/Cost-of-a-Data-Breach-Report-2024.pdf.

[56] Intel. 2023. Intel TDX module 1.0 specification. https://cdrdv2.intel.com/v1/dl/getContent/733568(2023).

[57] David Kaplan. 2017. Protecting VM Register State With SEV-ES. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/Protecting-VM-Register-State-with-SEV-ES.pdf.

[58] Vladimir Kiriansky and Carl A. Waldspurger. 2018. Speculative Buffer Overflows: Attacks and Defenses. *CoRR* abs/1807.03757 (2018). arXiv:1807.03757 http://arxiv.org/abs/1807.03757

[59] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. 1–19. https://doi.org/10.1109/SP.2019.00002

[60] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings (Lecture Notes in Computer Science)*, Neal Koblitz (Ed.). 104–113. https://doi.org/10.1007/3-540-68697-5_9

[61] Dayeol Lee, Kevin Cheang, Alexander Thomas, Catherine Lu, Pranav Gaddamadugu, Anjo Vahldiek-Oberwagner, Mona Vij, Dawn Song, Sanjit A. Seshia, and Krste Asanovic. 2022. Cerberus: A Formal Approach to Secure and Efficient Enclave Memory Sharing. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). 1871–1885. https://doi.org/10.1145/3548606.3560595

[62] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, Engin Kirda and Thomas Ristenpart (Eds.). 557–574.

[63] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. 2019. Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 1257–1272. https://www.usenix.org/conference/usenixsecurity19/presentation/li-mengyuan

[64] Xiang Li, Fabing Li, and Mingyu Gao. 2023. FLARE: A Fast, Secure, and Memory-Efficient Distributed Analytics Framework (Flavor: Systems). *Proc. VLDB Endow.* 16, 6 (2023), 1439–1452. https://doi.org/10.14778/3583140.3583158

[65] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter R. Pietzuch. 2017. Glamdring: Automatic Application Partitioning for Intel SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, Dilma Da Silva and Bryan Ford (Eds.). USENIX Association, 285–298. https://www.usenix.org/conference/atc17/technical-sessions/presentation/lind

[66] Linux. 2019. Linux vsock address family. https://manpages.ubuntu.com/manpages/jammy/man7/vsock.7.html.

[67] Linux. 2021. Multipurpose relay (SOcket CAT). https://manpages.ubuntu.com/manpages/jammy/man7/vsock.7.html.

[68] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. 605–622. https://doi.org/10.1109/SP.2015.43

[69] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. 2022. A Survey of Microarchitectural Side-channel Vulnerabilities, Attacks, and Defenses in Cryptography. *ACM Comput. Surv.* (2022), 122:1–122:37. https://doi.org/10.1145/3456629

[70] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. 2007. *Power analysis attacks - revealing the secrets of smart cards*. Springer.

[71] Shamiek Mangipudi, Pavel Chuprikov, Patrick Eugster, Malte Viering, and Savvas Savvides. 2023. Generalized Policy-Based Noninterference for Efficient Confidentiality-Preservation. *Proc. ACM Program. Lang.*, Article 117 (2023), 25 pages. https://doi.org/10.1145/3591231

[72] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David M. Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. 2017. ROTE: Rollback Protection for Trusted Execution. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, Engin Kirda and Thomas Ristenpart (Eds.). 1289–1306. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/matetic

[73] David A. McGrew and John Viega. 2004. The Security and Performance of the Galois/Counter Mode (GCM) of Operation. In *Progress in Cryptology - INDOCRYPT 2004, 5th International Conference on Cryptology in India, Chennai, India, December 20-22, 2004, Proceedings (Lecture Notes in Computer Science)*, Anne Canteaut and Kapalee Viswanathan (Eds.), Vol. 3348. Springer, 343–355. https://doi.org/10.1007/978-3-540-30556-9_27

[74] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos V. Rozas. 2016. Intel® Software Guard Extensions (Intel® SGX) Support for Dynamic Memory Management Inside an Enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016, HASP@ICSA 2016, Seoul, Republic of Korea, June 18, 2016*. 10:1–10:9. https://doi.org/10.1145/2948618.2954331

[75] Masanori Misono, Dimitrios Stavrakakis, Nuno Santos, and Pramod Bhatotia. 2024. Confidential VMs Explained: An Empirical Analysis of AMD SEV-SNP and Intel TDX. *Proc. ACM Meas. Anal. Comput. Syst.* 8, 3 (2024), 36:1–36:42. https://doi.org/10.1145/3700418

[76] Saeid Mofrad, Fengwei Zhang, Shiyong Lu, and Weidong Shi. 2018. A comparison study of intel SGX and AMD memory encryption technology. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy, HASP@ISCA 2018, Los Angeles, CA, USA, June 02-02, 2018*, Jakub Szefer, Weidong Shi, and Ruby B. Lee (Eds.). ACM, 9:1–9:8. https://doi.

org/10.1145/3214292.3214301
[77] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. CacheZoom: How SGX Amplifies the Power of Cache Attacks. In *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings (Lecture Notes in Computer Science)*, Wieland Fischer and Naofumi Homma (Eds.). 69–90. https://doi.org/10.1007/978-3-319-66787-4_4
[78] P. Mohan, A. Thakurta, E. Shi, D. Song, and D. Culler. 2012. GUPT: Privacy Preserving Data Analysis Made Easy. In *ACM SIGMOD International Conference on Management of Data (SIGMOD'12)*. 349–360.
[79] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. 2018. SEVered: Subverting AMD's Virtual Machine Encryption. In *Proceedings of the 11th European Workshop on Systems Security, EuroSec@EuroSys 2018, Porto, Portugal, April 23, 2018*, Angelos Stavrou and Konrad Rieck (Eds.). 1:1–1:6. https://doi.org/10.1145/3193111.3193112
[80] K. Morris. 2021. *Infrastructure as Code: Dynamic Systems for the Cloud Age* (2 ed.). O'Reilly.
[81] Kit Murdock, David F. Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. 1466–1482. https://doi.org/10.1109/SP40000.2020.00057
[82] OPAQUE. 2023. OPAQUE. The Confidential AI Company. https://www.opaque.co/.
[83] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. 3860 (2006), 1–20. https://doi.org/10.1007/11605805_1
[84] Pascal Paillier. 1999. Public-key Cryptosystems Based on Composite Degree Residuosity Classes. 223–238. http://dl.acm.org/citation.cfm?id=1756123.1756146
[85] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinarayanan. 2016. Big Data Analytics over Encrypted Datasets with Seabed. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 587–602. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/papadimitriou
[86] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2012. CryptDB: Processing Queries on an Encrypted Database. *Commun. ACM* 55, 9 (Sept. 2012), 103–111. https://doi.org/10.1145/2330667.2330691
[87] Davy Preuveneers and Wouter Joosen. 2015. SparkXS: Efficient Access Control for Intelligent and Large-Scale Streaming Data Applications. In *2015 International Conference on Intelligent Environments*. 96–103.
[88] Do Le Quoc, Franz Gregor, Jatinder Singh, and Christof Fetzer. 2019. SGX-PySpark: Secure Distributed Data Analytics. In *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, Ling Liu, Ryen W. White, Amin Mantrach, Fabrizio Silvestri, Julian J. McAuley, Ricardo Baeza-Yates, and Leila Zia (Eds.). ACM, 3564–3563. https://doi.org/10.1145/3308558.3314129
[89] Ronald L Rivest, Len Adleman, Michael L Dertouzos, et al. 1978. On data banks and privacy homomorphisms. *Foundations of secure computation* (1978).
[90] Indrajit Roy, Srinath T. V. Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. 2010. Airavat: Security and Privacy for MapReduce. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2010, April 28-30, 2010, San Jose, CA, USA*. USENIX Association, 297–312. http://www.usenix.org/events/nsdi10/tech/full_papers/roy.pdf
[91] B. Sang, P.-L. Roman, P. Eugster, H. Lu, S. Ravi, and G. Petri. 2020. *PLASMA: Programmable Elasticity for Stateful Cloud Computing Applications*. Vol. 42. 1–15 pages.
[92] Savvas Savvides, Darshika Khandelwal, and Patrick Eugster. 2020. Efficient Confidentiality-Preserving Data Analytics over Symmetrically Encrypted Datasets. *Proc. VLDB Endow.* 13, 8 (2020), 1290–1303. https://doi.org/10.14778/3389133.3389144
[93] Savvas Savvides, Julian James Stephen, Masoud Saeida Ardekani, Vinaitheerthan Sundaram, and Patrick Eugster. 2017. Secure Data Types: A Simple Abstraction for Confidentiality-preserving Data Analytics *(SoCC '17)*. ACM, New York, NY, USA, 479–492. https://doi.org/10.1145/3127479.3129256
[94] 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 *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 38–54.
[95] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). 753–768. https://doi.org/10.1145/3319535.3354252
[96] 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 *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. 955–970. https://doi.org/10.1145/3373376.3378469
[97] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. 2017. Panoply: Low-TCB Linux Applications With SGX Enclaves. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/panoply-low-tcb-linux-applications-sgx-enclaves/
[98] Simon Johnson, Raghunandan Makaram, Amy Santoni, and Vinnie Scarlata. 2021. Supporting intel sgx on multi-socket platforms. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/supporting-intel-sgx-on-mulit-socket-platforms.pdf.
[99] Jakub Szefer. 2019. Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses. *J. Hardw. Syst. Secur.* (2019), 219–234. https://doi.org/10.1007/S41635-018-0046-1
[100] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. 2017. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, Engin Kirda and Thomas Ristenpart (Eds.). 1057–1074. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang
[101] Shruti Tople, Shweta Shinde, Zhaofeng Chen, and Prateek Saxena. 2013. AUTOCRYPT: enabling homomorphic computation on servers to protect sensitive web content. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*. 1297–1310. https://doi.org/10.1145/2508859.2516666
[102] TPC. 1988. TPC-H benchmark. http://www.tpc.org/tpch/.
[103] 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 (ATC '17)*. 645–658.
[104] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. 2013. Processing Analytical Queries over Encrypted Data. 6, 5 (2013), 289–300. http://www.vldb.org/pvldb/vol6/p289-tu.pdf
[105] UC Berkley RISE Lab. 2021. MC2. https://mc2-project.github.io/opaque-sql-docs/src/benchmarking/benchmarking.html.
[106] Huseyin Ulusoy, Pietro Colombo, Elena Ferrari, Murat Kantarcioglu, and Erman Pattuk. 2015. GuardMR: Fine-grained Security Policy Enforcement for MapReduce Systems. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, Singapore, April 14-17, 2015*, Feng Bao, Steven Miller, Jianying Zhou, and Gail-Joon Ahn (Eds.). 285–296. https://doi.org/10.1145/2714576.2714624
[107] Huseyin Ulusoy, Murat Kantarcioglu, Erman Pattuk, and Kevin W. Hamlen. 2014. Vigiles: Fine-Grained Access Control for MapReduce Systems. In *2014 IEEE International Congress on Big Data, Anchorage, AK, USA, June 27 - July 2, 2014*. 40–47. https://doi.org/10.1109/BIGDATA.CONGRESS.2014.16
[108] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-Flight Data Load. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. 88–105. https://doi.org/10.1109/SP.2019.00087
[109] S. van Schaik, A. Seto, T. Yurek, A. Batori, B. AlBassam, D. Genkin, A. Miller, E. Ronen, Y. Yarom, and C. Garman. 2024. SoK: SGX.Fail: How Stuff Gets eXposed. In *2024 IEEE Symposium on Security and Privacy (SP)*. 248–248. https://doi.org/10.1109/SP54263.2024.00260
[110] Veracruz Contributors. 2024. Veracruz: Confidential Collaborative Computation. https://veracruz.readthedocs.io/en/latest/.
[111] Bin Cedric Xing, Mark Shanahan, and Rebekah Leslie-Hurd. 2016. Intel® Software Guard Extensions (Intel® SGX) Software Support for Dynamic Memory Allocation inside an Enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016, HASP@ICSA 2016, Seoul, Republic of Korea, June 18, 2016*. 11:1–11:9. https://doi.org/10.1145/2948618.2954330
[112] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 640–656. https://doi.org/10.1109/SP.2015.45
[113] Tao Xue, Yu Wen, Bo Luo, Gang Li, Yingjiu Li, Boyang Zhang, Yang Zheng, Yanfei Hu, and Dan Meng. 2023. SparkAC: Fine-Grained Access Control in Spark for Secure Data Sharing and Analytics. *IEEE Trans. Dependable Secur. Comput.* 20, 2 (2023), 1104–1123. https://doi.org/10.1109/TDSC.2022.3149544
[114] Tao Xue, Yu Wen, Bo Luo, Boyang Zhang, Yang Zheng, Yanfei Hu, Yingjiu Li, Gang Li, and Dan Meng. 2020. GuardSpark++: Fine-Grained Purpose-Aware Access Control for Secure Data Sharing and Analysis in Spark. In *ACSAC '20: Annual Computer Security Applications Conference, Virtual Event / Austin, TX, USA, 7-11 December, 2020*. 582–596. https://doi.org/10.1145/3427228.3427640
[115] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, Kevin Fu and Jaeyeon Jung (Eds.). 719–732. https://www.usenix.org/conference/usenixsecurity14/

technical-sessions/presentation/yarom

[116] 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 *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, Steven D. Gribble and Dina Katabi (Eds.). USENIX Association, 15–28. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[117] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y. Thomas Hou. 2018. TruSense: Information Leakage from TrustZone. In *2018 IEEE Conference on Computer Communications, INFOCOM 2018, Honolulu, HI, USA, April 16-19, 2018*. 1097–1105. https://doi.org/10.1109/INFOCOM.2018.8486293

[118] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y. Thomas Hou. 2018. TruSense: Information Leakage from TrustZone. In *2018 IEEE Conference on Computer Communications, INFOCOM 2018, Honolulu, HI, USA, April 16-19, 2018*. 1097–1105. https://doi.org/10.1109/INFOCOM.2018.8486293

[119] Xuyang Zhao, Mingyu Li, Erhu Feng, and Yubin Xia. 2022. Towards A Secure Joint Cloud With Confidential Computing. In *2022 IEEE International Conference on Joint Cloud Computing (JCC)*. 79–88. https://doi.org/10.1109/JCC56315.2022.00019

[120] 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 *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 283–298. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zheng

[121] Wenting Zheng, Ryan Deng, Weikeng Chen, Raluca Ada Popa, Aurojit Panda, and Ion Stoica. 2021. Cerebro: A Platform for Multi-Party Cryptographic Collaborative Learning. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael D. Bailey and Rachel Greenstadt (Eds.). USENIX Association, 2723–2740. https://www.usenix.org/conference/usenixsecurity21/presentation/zheng