Security Policy as Code

Pavel Chuprikov, Patrick Eugster, and Shamiek Mangipudi | Università Della Svizzera Italiana

Engineering software systems to fulfill security requirements remains challenging. We advocate for designing and implementing software systems around integrated advanced security policies, capturing security requirements. We report on experience gathered with this approach in confidentiality preserving data analytics.

Software systems continue to cause security concerns, particularly with respect to the confidentiality of data that they handle. Data are commonly referred to as the new currency, used in all walks of life for making more informed decisions guided by computations on large datasets. With data breaches remaining common,[6] improving the security of software systems handling such data becomes crucial.

Introduction

We posit that an important cause for security concerns, and, eventually, issues, is the fact that security requirements and constraints are not sufficiently embedded into software systems.

Multilevel Security

Consider the well-known approach for capturing constraints, notably on the confidentiality of data in a differentiated manner, consisting of assigning levels or labels l to information or users with different security clearances (or, more generally, principals). By ensuring that the set of levels $\mathcal{L} = \{l_1, \ldots l_n\}$ is arranged according to a lattice inducing a (reflexive and transitive) partial ordering $\preccurlyeq$ and with top $\top$ (most secret) and bottom $\bot$ (unprotected) elements, there is always the possibility of aggregating over several levels of $\mathcal{L}$ by finding a level that conservatively "includes" all those levels.[2]

Such multilevel security (MLS) is notably used for controlling access to classified information. It can be complemented by a similar model for reasoning about data integrity, and the two can be combined.[10]

The Front-End Perspective

MLS classically deals with controlling the access of users to data with different classifications, and it has been the starting point for many other more refined access control models. Figure 1(a) shows a simple example of a lattice that can be used for controlling access to data that are correspondingly labeled, with an example of such data given in Figure 1(b).

This access control angle captures the "front-end" perspective of software systems, which governs "who" has access to "what." While such access models effectively capture security requirements, they are typically not fully followed through in terms of software.

Digital Object Identifier 10.1109/MSEC.2025.3535803
Date of publication 28 February 2025; date of current version: 17 March 2025

More precisely, while these models are not only used to document policies but are effectively used to automatically enforce such policies in access to systems, this enforcement occurs at a coarse granularity, and it does not capture "how" an access is to occur or "which" means are allowed for it.

The latter two questions reflect an equally important "back-end" aspect of secure computer systems, with increasing focus in the context of big data analytics. This overlooked aspect notably affects the design and development of secure software systems in the era of increasing execution of data processing applications across untrusted infrastructures, such as cloud and edge data centers.

> **Software systems continue to cause security concerns, particularly with respect to the confidentiality of data that they handle.**

In the following, we elaborate on this shortcoming and on our approach to address it, showcased through our hybrid approach to distributed confidentiality preserving data analytics (Hydra) system for confidentiality preserving cloud-based data analytics,[7] at the culmination of a decade of research on the topic (e.g., Stephen et al.,[14] Savvides et al.,[12] and Savvides et al.[11]).

Secure Big Data Processing

The continuously increasing use of large datasets for decision making has been made possible by the rise of third-party shared infrastructure, starting with shared resources in cloud data centers. More recently, these data centers have been complemented by edge data centers to reduce response times by performing parts of computations closer to data producers and consumers.

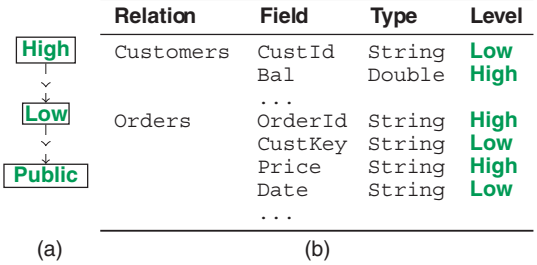

Figure 1. (a) A simple security lattice with three levels—`High` (= ⊤), `Low`, and `Public` (= ⊥)—and (b) extract of `Customers` and `Orders` relations of the popular TPC-H benchmark, with levels assigned.

Beyond Standard Encryption

This evolution toward computing in third-party infrastructure has accentuated the need for security in general and in particular for the protection of data while they are being computed on or with as well as data in use. This need clearly exceeds the capabilities of standard cryptographic schemes deployed for the storage and retrieval of data at rest. With our society becoming increasingly driven by the extraction of intelligence from data and large language models dominating the news, many (security) primitives and mechanisms have been proposed to secure data while they are subject to computation as well as to secure the computation itself. Examples include software-based primitives like homomorphic encryption (HE), including fully HE schemes[4] and partially HE (PHE) schemes (e.g., Paillier[9]), and hardware-based trusted execution environments (TEEs). Every major processor manufacturer, meanwhile, offers a TEE, e.g., Intel Software Guard Extensions (SGX), AMD Secure Encrypted Virtualization (SEV), and ARM Confidential Compute Architecture. Amazon Web Services (AWS), the largest cloud provider, has recently introduced its own product, Nitro. The advent of such TEEs, a natural and welcome evolution, has given rise to the moniker *confidential computing*. Big data processing can be seen as a subset of outsourced computation[13] with a strong focus on performance, notably scalability to large datasets, thus favoring certain mechanisms of over others.

Embarrassment of Riches

The large variety of such mechanisms leads to an embarrassment of riches, as the mechanisms exhibit strong differences along several dimensions, even within a same class (HE or TEEs):

- **Features**: Different mechanisms are used in quite different ways. For instance, PHE schemes support only individual operations, which might even be different from the operation that a programmer "logically" seeks to perform (e.g., addition in Paillier[9] requires ciphertexts to be multiplied). Certain schemes allow data with different keys to be combined, thus supporting multiparty computation, while standard PHE schemes require all combined data to be encrypted with the same key. TEEs typically offer the abstraction of memory enclaves—designating portions of

memory that are encrypted/decrypted automatically on the way from/to the processor—requiring code for explicitly setting them up (creation and attestation) and exchanging data with them (input/output). Despite such similar functionalities, programming interfaces, e.g. for instantiating enclaves or interacting with them, vary strongly between incarnations. Some TEEs can be used to deploy entire containers or even virtual machines; others are more resource constrained, making offerings very difficult to interchange.

- **Security**: Even only at a high level, mechanisms can offer different security guarantees like confidentiality -and/or integrity. For instance, (some) HE schemes support data integrity in the sense that tampering with ciphertexts can lead to errors at decryption, while TEEs can also achieve integrity of computation and attestation of the executed code. Even within such a guarantee (often rather informally specified), the mechanisms differ importantly in their definition of the guarantee and in the conditions under which it is achieved. An important question, of course, is with respect to what attacker or adversary model such guarantees are achieved. HE schemes and TEEs all have known attacks.[8] Note that while software solutions typically cannot be attested, they can usually be inspected by users quite easily. Hardware solutions, on the other hand, require certificates with a root of trust for attesting the code they execute and cannot themselves be easily inspected.
- **Performance**: Different mechanisms can exhibit vastly different performance characteristics. With TEEs benefitting from hardware support for encryption and decryption, these operations are usually very efficient in comparison to many HE schemes. Inversely, with actual operations on data protected with HE, certain schemes can be surprisingly effective by not requiring such de-/encryption at all. Beyond those differences, certain TEEs are more bound by memory constraints, others by computations or input/output. While TEEs are getting faster, so are HE schemes. Definitely, TEEs are currently not overhead-free, especially when used to secure entire containers or virtual machines transparently without modifications, as often promoted by manufacturers as part of pain-free deployment (which also negatively affects the trusted computing base). When used in distributed computations in untrusted clouds, data transferred between TEE instances also have to be encrypted/decrypted.
- **Deployment**: Clearly, not all cloud and edge data center providers support all, or even the same set of, TEEs. (At the time of writing, AWS supports only its own Nitro TEE, and Google Cloud Platform supports only SEV. SGX is present only in Microsoft Azure.) This disagreement is particularly problematic with computations involving data at or from both the edge and the cloud, as respective data centers are usually operated by different providers, supporting different mechanisms.

All the above differences slow adoption of corresponding mechanisms, which contributes to 82% of all data breaches these days involving data from the cloud.[6]

Security Policy as Code

The above differences make it pretty much impossible to identify a clear winning mechanism for all computations on all types of data across all environments.

The Back-End Perspective

The above observations raise the question of which security mechanism(s) to concretely use where and when. No solution may perfectly fit the requirements, which are obviously bounded by constraints of current solutions and may change over time. Strong security is obviously required and desired, but we have not yet converged on standardized solutions for data in use, as we have for data at rest.

Beyond all technicalities of the solutions themselves, even experts may choose differently at a given point in time, as they may assess the risks of an attacker successfully mounting a certain attack differently. Such an assessment can also depend on the technical environment (beyond the TEEs or HE schemes), including the hardware deployed and its configuration and the software stack supported, down to version numbers of individual software packages. Lastly, besides such platform differences, the choice of data center operator (cloud or edge provider) can also make an important difference, including through other protection mechanisms and attacker countermeasures deployed and even internal management processes. As some of the variables mentioned above may not be entirely known to experts on the outside, nonbinary risk assessment becomes necessary, leading to a more differentiated view of security, away from a binary static picture.

Extended Security Policy

Absolute security is impossible, and corporations have to put numbers on costs as well as revenues, encouraging the quantification of security risks reflecting the likelihoods of attacks materializing, compounded by estimated financial losses and weighed against costs for the acquisition and operation of protection and defense solutions. This weighing calls for a malleable notion of trust to capture the wide spectrum of security requirements.

In the spirit of MLS, we augment a lattice of security levels with a mapping of these levels to security mechanisms, including both hardware- and software-based ones (or combinations thereof), together with domains,

capturing a notion of sufficient trustworthiness. More precisely, with $\mathcal{M}$ and $\mathcal{D}$ denoting the set of available security mechanisms and domains, respectively, the extended security policy then revolves around tuples of the form

$$\langle m, d, l \rangle$$

where mechanism $m \in \mathcal{M}$ is deemed to be "strong or trustworthy enough" to protect data (or any principal) at level $l \in \mathcal{L}$ or below (according to $\prec$) in domain $d \in \mathcal{D}$. The domain d typically represents an infrastructure with a corresponding operator/provider. Separating it from the security mechanisms allows supporting same mechanisms in infrastructures run by different operators (e.g., both Microsoft Azure and Google Cloud offer SEV) which may not be necessarily prone to the same attacks.

The immediate understanding is that a system built around such a policy, for treating data at level l in domain d, can choose any mechanism assigned to l or a level l' above l ($l \prec l'$) for d. Figure 2(a) presents an example of a mapping for the lattice of Figure 1(a), and Figure 2(b) provides an example of how schemes can be inferred by a system for the relations of Figure 1(b) for a given query based on their assigned levels and the mapping.

Guarantee

Our Hydra[7] system, outlined shortly, leverages this extended security policy to provide a variant of noninterference[5] tailored to the MLS setup: no adversary capable of tampering only with mechanisms below (weaker than) some l ($\prec l$) can tell the difference between two evaluations of the same expression as part of a query, differing only in the input data at level l or above ($\succcurlyeq l$).

For simplicity, we focus here on confidentiality, while the approach can be used also for integrity, and the two can be combined.[10]

Mechanism ∈ $\mathcal{M}$	Domain ∈ $\mathcal{D}$	Level ∈ $\mathcal{L}$
_	Client	High
SGX, AES-GCM	Azure	High
SWP, AES-ECB, Paillier, ElGamal, OPE	Azure	Low

(a)

Field	Scheme
CustId	AES-ECB
Bal	AES-GCM
...	
OrderId	AES-GCM
CustKey	AES-ECB
Price	AES-GCM
Date	OPE
...	

(b)

Figure 2. An (a) example mapping for an extended security policy for the simple lattice of Figure 1(a) and (b) possible assignment of schemes for `Customers` and `Orders` relations inferred from the mapping based on the levels assigned in Figure 1(b). SGX and `Paillier` stand for the mechanisms introduced already, and the others except for AES variants are partially homomorphic (or property preserving) encryption schemes. AES: advanced encryption standard; GCM: Galois counter mode; ECB: electronic code book; OPE: order-preserving encryption; SWP: Song, Wagner, and Perrig.

Putting Security Policy as Code to Work

Toward correctly applying our extended security policy with a mapping of security levels to security mechanisms in a concrete system, we introduce several constraints. If correctly enforced, our policy brings many benefits with respect to the several dimensions along which security mechanisms and their application—and, thus, security requirements—can differ.

Constraints

The constraints below ensure the correct operation of a system using our extended security policy. These constraints can be easily verified on a given security policy at system start-up or reconfiguration:

- *Plaintext handling*: To allow for unprotected handling of data in some domains (e.g., where protection mechanisms outside of the system may be in place or trust is high due to other reasons), we introduce an "empty" mechanism denoted by an underscore (_).
- *Intradomain transmission*: As we are concerned with the more generic scenario of a distributed data processing system, it is important that data can be transferred between nodes in a secure manner. Thus, our policy requires a cryptographic scheme to be defined for every level l and every domain d to allow for encrypted transmission of data at level l between nodes within d. For simplicity we can allow a wildcard $*$ to designate all domains, allowing a single rule to be added for all domains. A default like the popular Advanced Encryption Standard (AES) scheme in Galois countermode can then be introduced through a simple preset rule ⟨`AES-GCM`, $*$, `High`⟩, for which any more specific rule, e.g., for a level $l \prec$ `High`, adds alternatives.
- *Interdomain transmission*: For handling secure transmission between domains, we introduce a predefined Internet domain `Inet`. Similar to intradomain transmission, we can easily add a default by using a preset rule ⟨`AES-GCM`, `Inet`, `High`⟩. Note that no rules for `Inet` should permit hardware mechanisms like TEEs, as the domain is strictly used for communication and not for computation.
- *Client*: We also introduce a predefined client (user) domain `Client`, in which all levels are permitted in plaintext without a mechanism, i.e., ⟨_, `Client`, `High`⟩. This domain allows for client-side completion of

computation but also for re-encryption, which can be used to deal with inherent limitations of PHE schemes and in lieu of manual "escape hatches" often used in practice to deal with "label creep" in complex computations. This phenomenon results from the outcome of a combination of n data items, with different levels $l_1, \ldots, l_n$ being treated as the highest level l_i $(i \in [1..n])$ among the constituents $(l_j \preccurlyeq l_i \ \forall j \in [1..n])$ or even as a higher one l $(l_j \prec l \, \forall j \in [1..n])$ if none is higher than all others (in the worst case, $l = \top$). Of course, a security policy could define several custom client domains, e.g., capable of handling different levels.

Benefits

Treating such a policy as an integral part of a system provides a number of concrete benefits mitigating issues caused by differences among mechanisms, as outlined below:

- **Features**: Focusing on our security policy allows considering abstract **functionality** rather than how exactly mechanism features are to be used to achieve that functionality. Moreover, the system has improved **maintainability** going forward, supporting the addition of new mechanisms, for the same functionalities or possibly beyond, to $\mathcal{M}$ (or inversely, restricting the use of those deemed inappropriate based on new findings).
- **Security**: Similarly, security guarantees can be provided with respect to the security policy rather than specific mechanisms, allowing for custom threat **taxonomies** for mechanisms to be accommodated. Furthermore, building a system around an artifact capturing security requirements has the clear advantage of forcing system developers to think of overall security as part of **design** rather than considering it as an afterthought, and it allows persisting requirements all the way through development to deployment.
- **Performance**: If different mechanisms are applicable in terms of security, according to the policy, for the level of some data to be processed, the system can choose the "best" mechanism based on some other more inherent properties of the data and the computation to be performed and an objective function. For instance, the system can optimize for expected **latency** for a given query, or it can make broader decisions in terms of (resource) **efficiency** based on momentary resource availability, including choosing slower mechanisms when the nodes with faster ones are temporarily overloaded.
- **Deployment**: If a system is capable of decoupling security mechanisms, it supports **portability** among mechanisms and, thus, (cloud) platforms; this can extend to efficient **interoperability** among platforms by taking federation into account rather than, e.g., first straightforwardly copying all data to one place. Note that these properties do not directly imply the easy addition of new mechanisms and, thus, maintainability.

System Architecture

We have built the Hydra[7] confidentiality preserving distributed data analytics system around our extended security policy, leveraging the benefits outlined to address challenges and, inversely, overcoming challenges to reap the benefits.

Overview

Hydra is a system for distributed confidentiality preserving cloud-based data analytics (Figure 3 displays its architecture). Its codebase extends the widely used Apache Spark system, with around 40,000 lines of code (LOC). Hydra can be used by data analysts through Spark's unmodified SparkSQL interface, providing a data flow model for queries based on operators in the style of Structured Query Language (SQL) operators/operations arranged according to a directed acyclic graph.

Integrating Mechanisms

A first major conceptual challenge consisted in defining appropriate abstractions for capturing a variety of mechanisms (**features–functionality**). Hydra currently supports HE encryption schemes and TEE-based enclaves. In a first step, we introduced programming interfaces that capture the basic features of these types of mechanisms (**deployment–portability** and **features–maintainability**), adjusted to the internal data processing

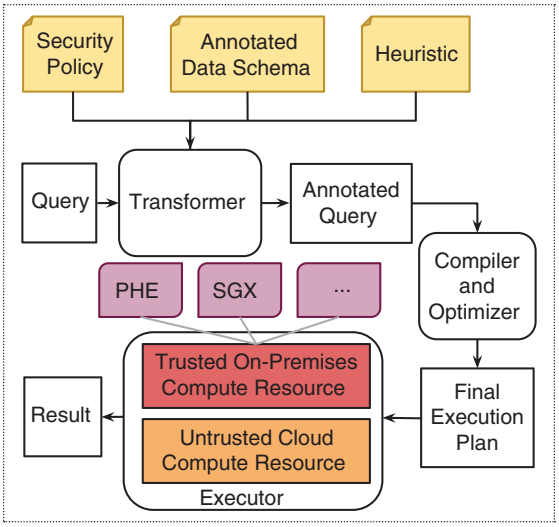

Figure 3. The architecture of Hydra built around our extended security policy. Except for the orange parts, execution happens on the client side, where the analyst resides.

model of Hydra (deployment–interoperability and **features–maintainability**). Interfaces for HE schemes typically include functions for encryption and decryption and for homomorphic operations they support. For TEEs, we have different levels of support—native, container, and virtual machine—with respective interfaces. We have reaped the maintainability benefits (**features–maintainability**) of our approach when integrating SEV and Nitro, with very limited effort. More specifically, integrating these into Hydra required only two and three new LOC, respectively, to be added (essentially for implementing a Scala class/trait). Integrating the first five PHE schemes required only a total of 76 LOC (22, 20, 14, 12, and eight LOC, respectively).

Runtime System

Spark has a complex malleable pipeline for query preprocessing preceding actual query execution. However, this part of the system is deployed on the trusted client side, and the extensions required for our purpose (see Figure 3) were beyond those foreseen for query optimization.

Spark also has a complex distributed runtime system, allowing it to scale to large datasets while making good use of data center resources and also allowing it to recover from process failures. The entire Hydra system being designed around our security policy, though extending an existing system, made it natural to consider enforcement of security requirements in the entire runtime system (**security–design**).

Selecting Mechanisms

To effectively achieve the potential benefits of our approach, notably in terms of **performance**, we introduced a notion of pluggable transformation heuristics for transforming SparkSQL queries, for example, to automatically use security mechanisms in a way that optimizes for a given objective function. These heuristics are similarly based on appropriate programming interfaces with, additionally, a domain-specific language for very easy description of simple heuristics, such as the one for the SGX-PHE hybrid execution used for Figure 4, comparing running times of Hydra to prior monolithic systems, which it easily outperforms (**performance–latency**). Dynamic decisions are possible in a limited form (**performance–efficiency**), with more advanced support being the subject of ongoing efforts. Note that our current heuristics framework enforces certain properties automatically by design (**security–design**), while others are verified on transformed queries based on a type system with respect to the security policy (**security–taxonomies**) enforcing our novel MLS noninterference guarantee.

Extensions

Many extensions to the basic mapping between security mechanisms and levels introduced earlier are possible. We first discuss potential extensions to our security policy to improve **security** and/or **performance** and tradeoffs within and between them and then discuss a potential extension to multiparty scenarios.

Options and Priorities

There are different ways of integrating a given security mechanism into a system; e.g., TEEs can be integrated to execute entire virtual machines and containers but also more natively specialized for the system at hand and its operations (**features–functionality**). For example, in Hydra,[7] SGX was supported originally through a custom-built interpreter, due to strict resource constraints and to enable the reuse of enclaves. At the policy level, this distinction can be currently folded into mechanisms themselves; e.g., $m_1 = \texttt{Nitro-dock}$, $m_2 = \texttt{Nitro-VM}$. The mappings of the form $\langle m, d, l \rangle$ can be extended with a v representing mode or variant to $\langle m, v, d, l \rangle$ in order to capture refined **security–taxonomies**.

While **performance** can (also) benefit when several mechanisms $m_1, \ldots, m_n$ are admissible for a same domain-level pair, i.e., the policy contains $\langle m_i, d, l \rangle$ $\forall i \in [1..n]$, it may be difficult for a system to entirely automatically choose the best option for a given case. A simple extension could, thus, consist in allowing

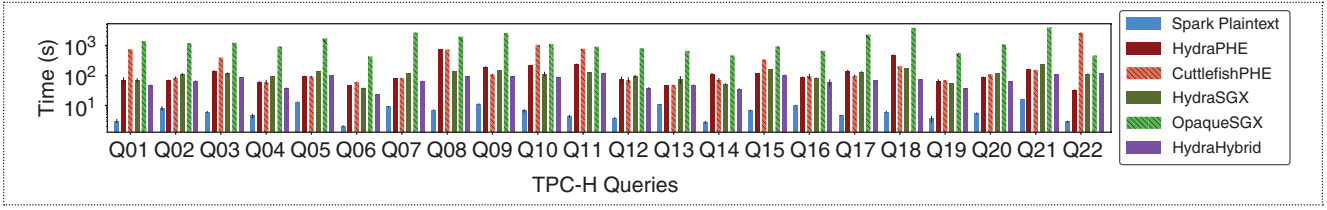

Figure 4. Performance benefits of combining mechanisms demonstrated by Hydra.[7] Latency (logarithmic scale) compared to Cuttlefish[12] and Opaque[15] on the TPC-H benchmark. Using only PHE, HydraPHE is on average 1.6× faster than Cuttlefish, and HydraSGX is 11.3× faster than Opaque. HydraHybrid, combining PHE and SGX, is 1.7× and 1.6× faster than HydraPHE and HydraSGX, respectively, and 2.7× and 17.9× faster than Cuttlefish and Opaque, respectively. HydraHybrid's simple heuristic leverages microbenchmarks of operators with different mechanisms to decide when to favor PHE over SGX.

mappings to include simple priorities for each option, e.g., $\langle m, d, l, p \rangle$, where p is an integer value. (This could be combined with the extension above.)

A more advanced approach could consist in identifying particular classes of scenarios, e.g., scenarios bound by memory, the central processing unit, or the input/output, and specifying priorities among the available options for each class separately.

Focusing on big data processing scenarios, one can consider integrating several measures related to data, inspired by the "five *V*'s,"[1] by extending security policies:

- *V1—volume*: Different mechanisms may be preferable for differently sized datasets, e.g., based on different initial one-time setup costs for mechanisms, like (homomorphic) encryption of data or enclave creation.
- *V2—velocity*: In some scenarios, **latency** may dominate (for individual queries), while for others, **efficiency** my dominate (with respect to many queries).
- *V3—value*: Beyond a possible implicit notion of value already captured by security levels, there are different means of assigning value to data that can be further used to set priorities (e.g., freshness).
- *V4—variety*: Data are a priori assigned levels independent of their data types. Especially complex data types can, however, favor certain mechanisms for better **performance**.
- *V5—veracity*: Within the same level, different mechanisms may be prioritized for a given data item based on the amount of data that was aggregated to determine it, e.g., to improve **performance** without a major downside to **security**.

Multiparty Scenarios

Another possible form of variety (see V4) can arise from multiparty scenarios with different organizations or stakeholders contributing data. This can even happen with suborganizations within larger corporations that grew organically with different divisions or possibly by acquiring and incorporating other corporations. There are two main ways in which extended security policies can handle such scenarios.

In the simplest case, the parties share one common security policy, and data are tagged with ownership information taken into account by the system at runtime. After introducing one or several auxiliary client domains with corresponding rules in $\mathcal{L}$ to access combined data, a system could then handle such data by considering for a given security mechanism whether it can handle data that are encrypted using different keys. This is typically the case for all common TEEs (assuming keys are accordingly managed and shared) and even some HE schemes. While a framework allowing mechanisms to be plugged in should provide appropriate interfaces for mechanisms to delineate their capabilities, like which operations are supported and how, in the case of (P)HE schemes, or the ability to support user-defined functions, such multiparty processing can also be added. This could also open the door to adding other mechanisms for multiparty processing, like garbled circuits.[13] Note that any of these feature categories could also be used in the mapping to help specify priorities.

For a more advanced form of multiparty setup with different parties having their own lattices $\mathcal{L}_i$, $i \in [1..n]$, a combined integrating lattice $\mathcal{L}$ can be created as a product lattice, just like when integrating confidentiality and integrity.[10]

The computing landscape is shifting its focus from controlling access to data to securing computations on such data, achieved by the means of third-party shared cloud and edge data centers.

With the vast majority of data breaches these days involving data in such infrastructures, it becomes important to secure corresponding computations and, thus, corresponding software systems.

Moving security policies capturing important security requirements from front-end systems and documentation to back-end systems, by having the latter systems built around such policies, has many benefits.

Many issues remain to be further addressed to complete our generic vision of "security policy as code" based on our extended notion of security policy with mappings to mechanisms and systems:

- *Securing the policy*: Governing the system, the security policy itself has to be secured. The easiest approach is to handle it as data with the highest confidentiality level $\top$, although in this context, clearly, integrity is almost more important than confidentiality to not offer attackers an easy target—modifying the mapping allows trivially bypassing all security mechanisms and measures.
- *Guarantees*: As mentioned, our extended security policy is not a priori limited to confidentiality. But subsumption in lattices for integrity usually follows the opposite direction than in the case of confidentiality (write down, read up for integrity versus write up, read down for confidentiality[10]), which can restrict the options for a system. Note that other guarantees, such as availability, which are also important for real-life systems and strongly impact software design and implementation, also need to be investigated.
- *Key management*: For simplicity, we have neglected the issue of key management. Like the security policy, keys also need to be protected. The simplest approach here is similarly to treat them as $\top$, but it may be

sufficient to label a key for encryption at level l with that same level.

- *Attestation*: Remote attestation of TEEs is important to ensure that they execute the intended code. Correct and efficient attestation brings its own challenges, which are likely to get accentuated by integrating different products.
- *Formalization*: Defining precise guarantees of a system orchestrated around a possibly changing security policy is, of course, as important as it is challenging. For Hydra and its focus on confidentiality, we defined a variant of noninterference. For other security guarantees, other frameworks might be more suited. A particularly interesting direction could be to investigate the application of differential privacy[3] in an MLS setup, particularly in the data processing context, due to the paradigm's connection to veracity (V5).
- *Policy transitions*: A system ideally allows its security policy to be changed on the fly, with immediate adaptation. Such changes are certainly simpler for a system like Hydra for online analytical processing with read-only queries, which can each be treated in isolation. For a more complex data processing system (e.g., storage), consistency would have to be considered more closely and defined more precisely for transitioning between two security policy versions.
- *Transparency*: Hydra manages to avoid escape hatches by resorting to client-side query completion or query re-encryption without visible performance degradation, allowing it to use and mix mechanisms transparently to users. Other systems may have to compromise on such transparency to reach performance goals or vice versa.
- *Generality*: Hydra currently supports HE and TEEs (with additional PHE schemes, like our highly efficient symmetric schemes[11] and Intel's novel Trust Domain Extensions TEE, currently being integrated), but other security mechanisms have been described with quite different features and guarantees. Integrating further mechanisms into Hydra, or into alternative data processing systems, will likely strongly affect architecture and programming interfaces and is likely to benefit from formal modeling. ■

Acknowledgment

We would like to thank Gerald Prendi, Malte Viering, and Savvas Savvides for their contributions to the Hydra project. This work was supported in part by U.S. National Science Foundation Grant 1618923, Swiss National Science Foundation Grants 192121 and 197353, Hasler Foundation Grant 20053, Cisco Research University Funding Grant 2853380, Meta Security Research Grant 474960397718052, and the Northrop Grumman Cybersecurity Research Center. Patrick Eugster is the corresponding author.

References

1. L. Husamaldin and N. Saeed, "Big data analytics correlation taxonomy," *Information*, vol. 11, no. 1, 2019, Art. no. 17, doi: 10.3390/info11010017.
2. D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, no. 5, pp. 236–243, May 1976, doi: 10.1145/360051.360056.
3. C. Dwork, F. McSherry, K. Nissim, and A. Smith, "Calibrating noise to sensitivity in private data analysis," *J. Privacy Confidentiality*, vol. 7, no. 3, pp. 17–51, 2017, doi: 10.29012/jpc.v7i3.405.
4. C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proc. ACM Symp. Theory Comput. (STOC)*, pp. 169–178, 2009, doi: 10.1145/1536414.1536440.
5. J. A. Goguen and J. Meseguer, "Security policies and security models," in *Proc. IEEE Symp. Secur. Privacy (S&P)*, pp. 11–20, 1982, doi: 10.1109/SP.1982.10014.
6. "Cost of a data breach report," IBM, Armonk, New York, USA, 2023. [Online]. Available: https://www.ibm.com/reports/data-breach
7. S. Mangipudi, P. Chuprikov, P. Eugster, M. Viering, and S. Savvides, "Generalized policy-based noninterference for efficient confidentiality-preservation," in *Proc. 44th ACM Int. Conf. Program. Lang. Des. Implementation (PLDI)*, pp. 267–291, 2023, doi: 10.1145/3591231.
8. A. Muñoz, R. Ríos, R. Román, and J. López, "A survey on the (in)security of trusted execution environments," *Comput. Secur.*, vol. 129, pp. 103–180, Jun. 2023, doi: 10.1016/j.cose.2023.103180.
9. P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Proc. 18th Annu. Int. Conf. Theory Appl. Cryptographic Techn. (EUROCRYPT)*, pp. 223–238, 1999.
10. R. S. Sandhu, "Lattice-based access control models," *IEEE Comput.*, vol. 23, no. 11, pp. 9–19, Nov. 1993, doi: 10.1109/2.241422.
11. S. Savvides, D. Khandelwal, and P. Eugster, "Efficient confidentiality-preserving data analytics over symmetrically encrypted datasets," in *Proc. VLDB Endowment*, 2020, vol. 13, no. 8, pp. 1290–1303, doi: 10.14778/3389133.3389144.
12. S. Savvides, J. J. Stephen, M. S. Ardekani, V. Sundaram, and P. Eugster, "Secure data types: A simple abstraction for confidentiality-preserving data analytics," in *Proc. ACM Symp. Cloud Comput. (SoCC)*, pp. 479–492, 2017.
13. Z. Shan, K. Ren, M. Blanton, and C. Wang, "Practical secure computation outsourcing: A survey," *ACM Comput. Surv.*, vol. 51, no. 2, pp. 1–40, Feb. 2018, doi: 10.1145/3158363.
14. J. J. Stephen, S. Savvides, R. Seidel, and P. Eugster, "Practical confidentiality preserving big data analysis," in *Proc. 6th USENIX Workshop Hot Topics Cloud Comput. (HotCloud)*, 2014.
15. W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An oblivious and encrypted

distributed analytics platform," in *Proc. USENIX Symp. Networked Syst. Des. Implementation (NSDI)*, pp. 283–298, 2017.

Pavel Chuprikov is an assistant professor at Télécom Paris, 91123 Palaiseau, France, and formerly a postdoctoral researcher in the Software Systems group, Università Della Svizzera Italiana 6900 Lugano, Switzerland. His research interests include the specification and optimization of distributed systems, with a particular focus on algorithms, programming abstractions, and network infrastructure. Chuprikov defended his Ph.D. in computer science at HSE University. Contact him at pavel.chuprikov@usi.ch.

Patrick Eugster is a full professor of computer science at the Università della Svizzera Italiana, 6900 Lugano, Switzerland. Where he leads the Software Systems group. His research interests include distributed systems, especially their intersection with security and programming languages. He is the recipient of a National Science Foundation CAREER award (2007) and a European Research Council Consolidator award (2012), and he is a former member of the DARPA Computer Science Study Panel (2011). Eugster holds a Ph.D. degree in computer science from EPFL. Contact him at eugstp@usi.ch.

Shamiek Mangipudi is a postdoctoral researcher in the Software Systems group, Università della Svizzera Italiana, 6900 Lugano, Switzerland. His research interests include scalable solutions backed by formal guarantees for secure cloud computing built on top of hardware- and software-based security mechanisms. Mangipudi received an M.Sc. in computer science from Purdue University and a Ph.D. degree in computer science from USI. Contact him at shamiekm@gmail.com.

Digital Object Identifier 10.1109/MSEC.2025.3548710