Styx: Collaborative and Private Data Processing With TEE-Enforced Sticky Policy

Shixuan Zhao
The Ohio State University
Columbus, OH, USA
zhao.3289@buckeyemail.osu.edu

Weicheng Wang
Purdue University
West Lafayette, IN, USA
wang3623@purdue.edu

Ninghui Li
Purdue University
West Lafayette, IN, USA
ninghui@purdue.edu

Zhiqiang Lin
The Ohio State University
Columbus, OH, USA
zlin@cse.ohio-state.edu

Abstract

Protecting sensitive information in data-driven collaborations, such as AI training, while meeting the diverse requirements of multiple mutually distrusted stakeholders, is both crucial and challenging. This paper presents Styx, a novel framework to address this challenge by integrating sticky policies with Trusted Execution Environments (TEEs). At a high level, Styx employs a hardware-TEE-protected middleware with a programming language runtime to form a sandboxed environment for both the data processing and policy enforcement. We carefully designed a data processing workflow and pipelines to enable a strong yet flexible data-specific policy enforcement throughout the entire data lifecycle and data derivation to achieve data-in-use protection, data lifecycle protection and dynamic collaboration. We implemented Styx and demonstrated its ability to make collaborative computing, such as joint AI training, more secure, privacy-preserving, and policy-compliant. Our evaluation shows the performance overheads imposed by Styx are reasonable on single-node computation with the capability to scale to a large distributed multi-node deployment.

CCS Concepts

• **Security and privacy → Domain-specific security and privacy architectures**; **Access control**; **Trusted computing**.

Keywords

TEE, Access Control, Middleware, Confidential Computing

ACM Reference Format:
Shixuan Zhao, Weicheng Wang, Ninghui Li, and Zhiqiang Lin. 2026. Styx: Collaborative and Private Data Processing With TEE-Enforced Sticky Policy. In *27th International Middleware Conference (Middleware '26), December 14–18, 2026, Tarragona, Spain.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3801927.3810466

1 Introduction

We study the problem of protecting sensitive data used in collaboration among multiple stakeholders, where no single party is universally trusted. In this setting, while stakeholders are performing the data processing in a collaborative fashion, stakeholders are mutually distrusted and may have different yet specific policies on how their sensitive data can be used, what can be included in the output as well as how the output needs to be protected. This requires strong enforcement of data policies from every stakeholder throughout the entire processing as well as guarantees on those generated data. Three critical requirements emerge in such scenarios: First, *data-in-use protection* must be achieved to ensure the computation itself does not violate the policies even if the computation is carried out by a party that is not universally trusted. Second, *data lifecycle protection* is needed to ensure that any output must be allowed by the stakeholders' policies and is protected with proper output policies. Third, *dynamic collaboration* must be supported to enable joint processing, ownership and policy decisions.

Sticky policies were introduced nearly two decades ago to support the denotation of the diverse policies of different stakeholders on different data by attaching data-specific policies to the data itself [39, 50]. While sticky policies help forming a scheme to denote the data-specific policies, existing works focus on building access control with sticky policies [48, 57, 60, 63], meaning that enforcement schemes must be redesigned to achieve the three requirements we want to achieve.

At a glance, confidential computing seems to offer a promising approach to achieve data-in-use protection. It isolates sensitive data in a Trusted Execution Environment (TEE) during processing [27, 53, 61, 66] and has been widely adopted in cloud services, such as AWS [1], Google Cloud Platform [29], and Microsoft Azure [69]. However, currently TEEs are primarily designed for outsourcing settings, where a single entity owns, processes, and stores the data. While TEEs offer remote attestation mechanisms to ensure the integrity of the applications even to multiple stakeholders, the attestation only conveys a hash and each stakeholder needs to verify the application before trusting the hash, which is particularly impractical in our setting where data can be processed by an untrusted third party using proprietary code. This means that TEEs fall short in providing data-in-use protection when the processing is done by other entities, not to mention data lifecycle protection in dynamic collaborations.

To address these limitations, we propose Styx, a middleware framework that combines the data-specific policies offered by sticky policies with the strong enforcement of TEEs, and extends

Middleware '26, December 14–18, 2026, Tarragona, Spain

ACM ISBN 979-8-4007-2621-7/26/12
https://doi.org/10.1145/3801927.3810466

with runtime sandboxing to achieve secure collaborative sensitive data processing among multiple stakeholders. However, designing such a secure framework presents several challenges raised by the three critical requirements we discussed. One primary challenge is ensuring data-in-use protection, as traditional encryption and access control methods are inadequate once data is being actively processed. Another significant challenge is to maintain data lifecycle protection, which requires policies to be enforced on the output side as well to specify not just what can be included in the output, but also what policies should be attached to the output just like the input data themselves. Besides, achieving dynamic collaboration involves consolidating different data policies from the input stakeholders and demands joint control and policies on the output. In Styx, we employed a runtime sandboxing design to ensure that the I/O of the data processing program is under full control of the middleware framework, denying any unauthorized leakage and ensuring the data-in-use protection. We also designed a workflow with dual pipelines on the input as well as the output side to maintain the data lifecycle protection. The pipelines are made to take data from different stakeholders with respect to each data's policy on both the input and the output for the dynamic collaboration. We use the same runtime to allow pluggable policy engines to support fully customizable policies.

We have implemented Styx into an architecture-independent framework and built a fully functional demo prototype upon it to demonstrate the feasibility and usability of this approach. The framework detached the protocol and framework from specific architectures, enabling data processing on heterogeneous processing nodes, which aligns with our vision for Styx in a distributed setting. The prototype of Styx is based on Intel SGX [83] as the TEE and WebAssembly (WASM) [7] as the runtime. We were able to successfully port real-world applications, such as libonnx, an ONNX engine, into the system. We also evaluated Styx using a motivating example involving multiple hospitals collaboratively training a cancer classification model. Our evaluation results showed that Styx imposed comparable overheads on data access to conventional encrypted protocols such as HTTPS, while the runtime sandboxing design's extra overheads were also minimal compared to standard runtime performance.

Contributions. We make the following contributions.

- **Sticky Policy and TEE Integration (§3):** We present our novel approach Styx that combines the flexibility of sticky policy with the strong execution guarantee of TEE. We extended the integration to achieve data-in-use protection, data lifecycle protection and dynamic collaboration using our carefully designed input output pipelines, runtime sandboxing, pluggable policy engine support, addressing the challenge of enforcing sticky policies for both usage and data derivation in collaborative environments.
- **Styx Framework and Prototype (§4):** We implemented Styx into an architecture-independent framework that contains the protocol and extendable interfaces. We built a fully functional prototype on Intel SGX with WASM, demonstrating the feasibility and practicality of Styx.
- **Application (§5) and Evaluation (§6):** We applied Styx to our motivating example to showcase a real-world application usage on how Styx can protect the sensitive data from multiple stakeholders. We evaluated on both real-world applications and synthetic workloads as well as large-scale simulation and demonstrated a reasonable overhead.

2 Existing Approaches

2.1 Cryptographic Approaches

Various cryptographic approaches have been developed to offer data-in-use protection. Secure Multiparty Computation (SMC) techniques enable multiple stakeholders to jointly perform a computation while ensuring that each party only sees their designated output. The theoretical foundations for SMC were established in the 1980s and 1990s, with the garbled circuits approach for the two-party case [78, 79] and the secret-sharing based approach for the multi-party case [22, 28]. These methods involve converting functions into boolean or arithmetic circuits then jointly evaluating them through a protocol. Recent research focused on improving the efficiency of these approaches [9, 18, 34, 46, 49, 75]. However, the performance overhead remains prohibitive for complex functions. Additionally, these methods offer limited protection for derived data. While they support different parties receiving different outputs, once an output is provided, other parties lose control over it even if their data was involved. Besides, requiring outputs to be encrypted using threshold cryptography significantly increases circuit size and overhead.

Homomorphic encryption [11, 21, 24, 25, 71] allows computations performed directly on ciphertext without revealing plaintext. Only the party with the private key can decrypt the result. This is beneficial for single-entity scenarios where data storage and computation are outsourced to the cloud. However, it does not support multiple stakeholders each providing inputs and jointly controlling outputs. Furthermore, homomorphic encryption remains prohibitively expensive for complex computations.

2.2 Access Control Approaches

Another approach to protect data-in-use involves access control. Sandhu et al. [77] introduced the concept of usage control, which considers complex access requirements during data use and obligations after data usage. Languages for usage control were developed in a series of papers [58, 59, 81], and enforcement of such policies was explored by augmenting SELinux MAC mechanisms [82]. Byzantine-tolerant can also be employed [26, 62].

Researchers also explored protecting data during computation and transmission. The notion of *sticky policies* was introduced almost two decades ago [39, 50] to help enterprises fulfill privacy policy commitments to end users. Sticky policies remain attached to data even as it moves across systems or networks. Recent surveys [48, 60] provide an overview of sticky policies. Similar ideas have also been introduced under *distributed usage control* [63] and *Policy-Carrying Data* (*PCD*) [57]. The enforcement of sticky policy is difficult. Some researchers explored enforcing sticky policies using encryption techniques such as Identity-Based Encryption [10], Attribute-Based Encryption [8, 30], and Proxy Re-encryption [31]. However, these techniques do not adequately support dynamic collaboration and derived data protection.

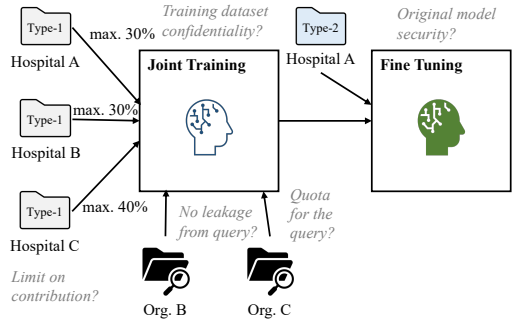

Figure 1: A motivating scenario that hospitals jointly train a cancer classification model, fine-tune the model, and query the model.

2.3 TEE Approaches

Trusted Execution Environments (TEEs) [17, 23, 51, 72] help protect data-in-use by creating secure, isolated environments that operate separately from the main operating system [6]. This isolation prevents unauthorized access or interference by other processes, ensuring data protection during the execution. Memory of TEEs is generally encrypted to maintain confidentiality, even if an attacker gains access to the system's physical memory [41].

Decisions about whether a program in the TEE can be trusted are done via remote attestations to verify the program's integrity and hardware genuineness, which offers no guarantees on the program's behavior, particularly on the input and output. Furthermore, TEEs face constraints due to limited input and output channels, posing challenges for collaborative computation. The software inside the TEEs is trusted universally after attestation, meaning that under a collaborative setting, one must fully trust the program to faithfully obey the policies blindly if the program cannot be formally verified or put under public inspection.

Contemporary commercial cloud services incorporate confidential computing features using hardware-based security technologies, including enclaves and secure virtual machines. Examples include Amazon AWS Nitro Enclaves [3] integrated with AWS Key Management Service [2], and Google Cloud Platform's confidential shielded VMs [16] and Key Management Service [15]. Hardware examples include Intel SGX [83] and AMD SEV [4]. These approaches support use cases where a single entity outsources computation to the cloud, instead of a collaborative computation design.

3 Architecture Design

3.1 Motivating Scenario

We first describe a scenario in which many organizations have sensitive data and would benefit from collaborating with each other to motivate our design requirements and design choices. For concreteness, we consider a medical use case in which multiple hospitals have collected diagnostic data from their patients and plan to jointly train a cancer classification model using machine learning (ML), as illustrated in Figure 1. To meet the regulation requirements and to protect patients' privacy, the diagnostic data must not be shared with each other. Note that information leakage can occur in several ways. First, information can potentially be leaked during training to the party controlling the software/hardware systems that run the training process. Second, information can be leaked through the model parameters, as information about the training data may be revealed through them. Third, information can be leaked through unlimited querying a model trained using the private diagnostic data and obtaining the result. This last form of leakage may be unavoidable and therefore needs to be controlled rather than eliminated.

We assume that hospitals have two types of diagnostic data, depending on their sensitivity and the regulations governing them. They are willing to contribute one type of data (type-1 data in Figure 1) for the joint training of the ML model in a way that their data is not revealed, and the trained model parameters are not revealed directly to other hospitals. They also have another type of diagnostic data (type-2 data) that is more sensitive than type-1. They do not want type-2 diagnostic data to be used in training any model that other hospitals can query. However, they want to use type-2 diagnostic data to fine-tune the jointly-trained model to obtain a model that only themselves can use.

In this setting, there is a jointly-trained base model. Some hospitals may not have any type-1 diagnostic data for training the base model, but are willing to pay money to either directly query the base model, or to fine-tune it using their type-2 diagnostic data. Some hospitals may have neither type-1 nor type-2 data and are willing to pay to query either the base model or some hospital's refined model. Furthermore, when a hospital queries an ML model, the hospital may require that the instances in the queries are not revealed to any other parties.

This example illustrates the three requirements we described:

Data-in-Use Protection. We need to ensure that data used in computation are not revealed. No one learns type-1 diagnostic data when training the base model. No hospital learns the base model parameters while fine-tuning the model with its own type-2 diagnostic data. A hospital submitting an instance to query a model does not learn anything other than the model prediction on the instance, and no other party learns anything about the instance.

Data Lifecycle Protection. We note that new data are being derived from computations involving protected data, and the derived data needs to be protected as well. For example, the base model is derived from type-1 diagnostic data, and fine-tuned models are derived from the base model and type-2 diagnostic data. The base model needs to be protected in a way that no hospitals can simply copy the model and use it in any way they want. Similar controls need to be applied to the fine-tuned model. The hospitals need to have confidence in these protections before the training can take place. However, it is challenging to support this.

Dynamic Collaboration. The hospitals have policies determining under what conditions their data can be provided to train the ML model, and how the ML models can be used in the future. For example, hospitals may not be providing equal amounts of training data and want their shared rights to control the model to be determined by the amount of data. Some hospitals may want to ensure that their share of data is sufficiently high so that they are able to exercise some higher level of control in the trained model. Some hospitals may want to ensure that their data do not constitute more than a

certain percentage of the total training data. We need mechanisms to combine such constraints to determine whether the joint computation can be carried out. Furthermore, not all usages for the derived data (ML models) may be known or can be determined at the time of training, and mechanisms are needed to update the policies later. For example, initially the models may be only used by hospitals providing training data, and later it may be decided that the model can be sold for revenue with some revenue sharing scheme.

3.2 Design Considerations

We propose the following high-level design to support the key requirements of data-in-use protection, data-life-cycle protection, and dynamic collaboration. Specifically, our observation is that data's lifecycle follows a producer-consumer pattern. A program that generates protected data is called a data *producer*. A program that reads and uses protected data is called a data *consumer*. Some programs can be both a data producer and a data consumer. Contents of protected data, attributes about the data, and policies governing how the data can be used are encrypted and packaged together. We call such data *Policy-Attached Data* (PAD).

However, ensuring the enforcement of the policy is not trivial. Sometimes a consumer's owner even has the motivation to peek at the data. There are three primary approaches to achieve this:

Black-Box Approach. A black-box approach requires the program to be verified and agreed by the stakeholders. The policy specifies which programs are allowed to access the protected data via hashes or certifications. The policy treats the program as a black-box and specifies nothing about the internal properties. This approach has the drawbacks of: (1) Has Limited support for dynamic collaboration since all stakeholders have to agree on the exact program; (2) Relies on program perfection for any protection to work; (3) Is impractical for complicated, diverse and proprietary consumer programs.

White-Box Approach. A white-box approach uses the policies to determine what computation actions can be performed on protected data. The consumer program is no longer a general purpose program but works like a script to operate those allowed actions on the data. Such approach may be suitable for simple applications with limited routines, e.g., when training ML models using standard techniques. However, for general-purpose apps, the routines can be difficult to specify. Meanwhile, verification of these trusted actions can also be heavy when the actions are complicated or massive. Besides, this approach requires extra yet hard-coded design to ensure the output is properly protected.

Gray-Box Approach (Sandboxing). This approach is a middle ground between the black-box approach and the white-box approach. The policy specifies constraints on the inputs and outputs of programs that can access protected data. To guarantee the enforcement of the policies, a special *middleware* running inside a TEE is employed. The middleware sandboxes the programs that use protected data, and controls all the I/O of the program. Before granting the program access, the middleware can verify that the policies of all provided input data are satisfied. When the program generates outputs, the middleware ensures that correct policies are attached to the output against the input's policy. We choose to adopt this gray-box approach, as we believe it offers the best trade-off in terms of feasibility, flexibility, and trust in the data-access programs.

3.3 Threat and Adversary Model

The "gray-box" approach allows us to define a very strong security model to protect the PADs. At high level, our threat model follows the same as common TEEs but with one significant difference: We consider data consumers to be untrusted. That is, other than those outside of the TEEs may want to steal the contents of PADs, the consumer program that processes the data may also want to leak the data or violate the policies attached to the data. We now discuss our threat model in details from external threats and internal threats.

External Threats. External threats refers to any adversaries outside the TEEs. They may get a PAD in the encrypted form but should not be allowed to read any portion of the data or the policy in the PAD in plaintext. We assume that TEEs are able to protect the code and data from external tampering to defend against these threats.

Internal Threats. Contrary to conventional TEE threat models which uniformly trust everything inside the TEE, our threat model divides the software inside the TEE into two domains: the untrusted consumer program domain and the trusted infrastructure domain. The consumer program is considered as untrusted which can be buggy or even malicious. However, the infrastructure domain where the middleware, runtimes and libraries lie is trusted. The infrastructure code should provide a sandboxed environment to host the consumer program domain. While the consumer program is free to perform any computation inside the sandboxed domain, its I/O is completely controlled by the infrastructure. Whenever the consumer program would like to read in or write out any sensitive data, it must go through policy checks to ensure that such I/O is allowed.

Our rationale is that unless being verified by a trusted third party or being open sourced for the community to inspect, a program cannot be considered as fully and universally trusted. Given that it is not practical to verify every version of every consumer program line by line and many consumer programs are proprietary, a consumer program naturally falls out of the trusted boundary.

Out-of-Scope Threats. Following common TEE threat models, this work does not address attacks against the TEE hardware stack (e.g., side-channel attacks [67] and speculative-execution attacks [42]), or malicious consumers leaking data content via covert channels (e.g., leaking data by encoding the data with CPU load).

3.4 Roles in Computation

In addition to data producer and consumer as discussed, we have also introduced two roles called *data custodian* and *key delegator* to address the data ownership and cryptographic requirements. We now discuss the details of these roles:

Data Custodian. A data custodian owns certain nodes in the system (e.g., producers and/or consumers). It represents the data owner(s), instructs the producer on specific policies to be attached to the data and provisions encryption keys. One does not have to expect the data custodian to always be online.

Key Delegator. A key delegator is a service that stores encryption keys for data custodians and provisions these keys to validated

```
⟨policy-attached-data⟩   ::= ⟨metadata⟩ ⟨encrypted-payload⟩
          ⟨metadata⟩     ::= ⟨data-id⟩ ⟨key-delegator-uri⟩
  ⟨plaintext-payload⟩    ::= ⟨raw-data⟩ ⟨policy⟩+ ⟨data-attribute⟩+
    ⟨data-attribute⟩     ::= ⟨attribute-name⟩ ⟨data-type-ref⟩ ⟨attribute-value⟩
          ⟨policy⟩       ::= ⟨policy-lang-id⟩ ⟨input-constraints⟩ ⟨program-constraints⟩ ⟨output-constraints⟩
```

Figure 2: A BNF specification for Pad. ⟨encrypted-payload⟩ is obtained from aes-encrypt$_{\text{datakey}}$(⟨plaintext-payload⟩).

producers and consumers. The only reason to have it is to help distribute the keys when the data custodian is not online and can be combined into the custodian as well. It is protected within a TEE and can be attested by the data custodian before being provisioned with the key. It will attest a target TEE before provisioning the encryption key to that TEE. Multiple instances can be launched and regular load balancing can be used for scalability.

Data Producer. A data producer is a TEE-protected program that generates and packs the raw data into Pad. It is provisioned with policies and encryption keys directly from the data custodian.

Data Consumer. A data consumer is a program that reads and operates on Pad. It is not allowed to access the data until the policy is checked and met. Even when it is granted with the access to the data under the policy, it must not leak any portion of the data directly. Its output should also comply with the original policy's restriction on the computation outputs.

3.5 The Elements of Policy-Attached Data

A BNF specification of policy-attached data (Pad) in Figure 2 to define how a Pad should be packed. To achieve the strong enforcement of the policy for a Pad, the Pad can only be accessed inside a TEE by a consumer after the policy is met. This requires most of the payload to be encrypted, leaving only minimal information in plaintext to help the TEE retrieve the key to decrypt the actual payload.

Metadata. The ⟨metadata⟩ is the only plaintext portion, providing information for retrieving the decryption key for a Pad.

Each Pad has exactly one data custodian who possesses the key used to encrypt the payload in Pad. The key is provisioned to the key delegator described in § 3.4 whose URI ⟨key-delegator-uri⟩ is included in the ⟨metadata⟩. Depending on the custodian's choice, the key can be data specific, and therefore a ⟨data-id⟩ is also included. When a consumer attempts to access the Pad, the trusted middleware can use these information to fetch the key from the key delegator after a successful attestation.

Payload. The ⟨plaintext-payload⟩ is a sequence of three elements: the *raw data*, one or more *policies* and the *data's attributes*. Intuitively, each policy specifies that the Pad can be read by a consumer program that complies with the policy written in the policy language of the given ⟨policy-lang-id⟩, and used as the input identified by ⟨input-constraints⟩, provided that the program satisfies the ⟨program-constraints⟩. The output of the consumer can be restricted with the ⟨output-constraints⟩, which can specify what is allowed in the output as well as what policy or ownership should be attached to the output. The ⟨data-attribute⟩ element is used to store information specific about the data that is used during the policy evaluation (e.g., time of creation, data count).

The payload is encrypted with an optional Cryptographic Message Authentic Code (CMAC) for integrity check before being appended to the metadata to form a Pad.

Policy Language. Each policy is written in a specific *policy language*, referred with the ⟨policy-lang-id⟩. The policy is to be interpreted by a policy engine implemented as a trusted pluggable module containing no data and can be made open for inspection. The engine makes Boolean decisions based on the input, the consumer program and the output against the policy. We discuss more about the policy engine in § 3.6.

3.6 Workflow and Components of Styx

The workflow and architecture of Styx must enforce the policy even if a consumer program is not trusted and can be compromised. We designed a complete workflow and interaction between the roles and components and illustrated them in Figure 3.

Preparation. First, the data custodian needs to specify the key delegator, the policies attached to the data, and data attributes used in policy verification before the data is created to the data producer. For some cases, the producer's infrastructure is also owned by the data custodian, and such an agreement can be hard-coded in the producer, e.g., an IoT device serves as a producer and the device owner is also the data custodian. If the producer's infrastructure is not directly owned by the data custodian, a secure communication is required before data is created.

Data Creation in the Producer. The producer first generates the raw data or obtains the raw data from trusted channels of the data custodian. After extracting data attributes needed by the policy, it then builds and encrypts the payload with a randomly generated symmetric data encryption key inside the TEE. This data encryption key is sent back to the data custodian who will provision it to the key delegator. The generated Pad can now be sent to the outside.

Data Computation in the Consumer. The critical protection needed in our design happens in the consumer that uses the data and derives new data based on the input. To solve the challenge that the consumer program is not considered as trusted, we propose to use runtime sandboxing technique to achieve a full control over the consumer program's I/O, forcing any I/O to go through our designed pipelines that enforce policy checks. This means that the TEE environment will run a consumer middleware, in which the consumer program is executed inside a language runtime. This middleware contains no proprietary logic and can be open for inspection for trust. There are four major components of the middleware:

- **Runtime.** The runtime serves as the sandboxed environment for the consumer. Technically, any runtime that requires I/O to be handled by the runtime can be used (e.g., Python, WebAssembly).

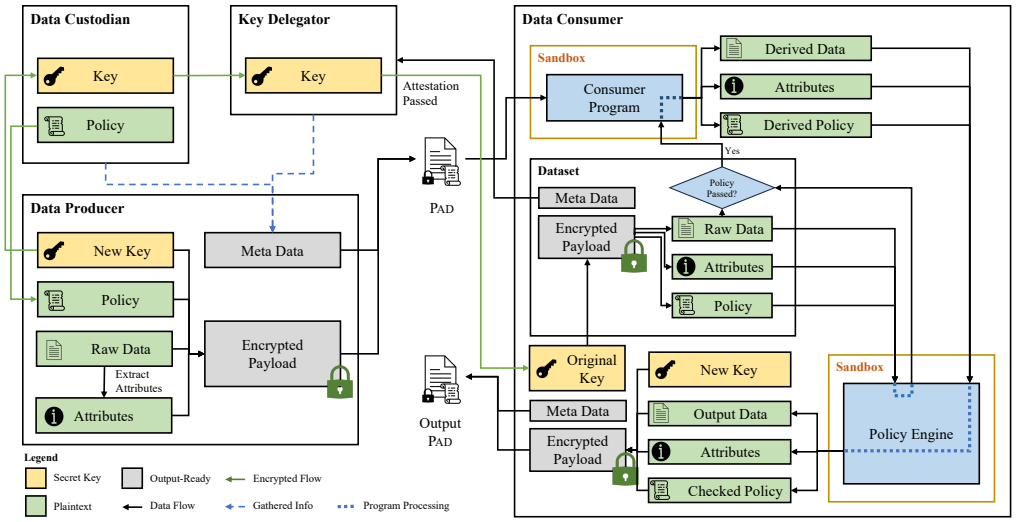

Figure 3: The workflow of the Styx.

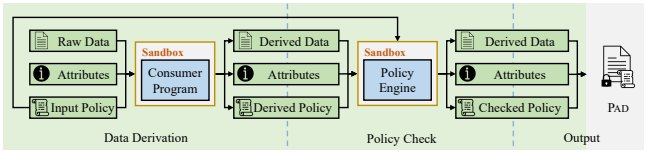

Figure 4: Data and policy derivation workflow. Green background means trusted domain. Gray background means untrusted domain (sandboxes or outside of TEE).

The consumer program is then written or compiled into the language or bytecode of the runtime for execution.

- **Styx Framework.** We designed our I/O pipelines and other supporting infrastructures into an architectural-independent framework library. The framework offers APIs to adapt to different runtimes, cryptographic algorithms as well as TEE interfaces.
- **Dataset.** Input PADs must form a *dataset* before a policy check. This is because policies can have inter-data dependencies such as percentage limit of one entity's data in the entire dataset.
- **Policy Engine Support.** As described in § 3.5, the policy engine is a trusted pluggable module that can support sophisticated policy languages to restrict the input, program and output. Essentially, this policy engine should be able to perform complex logics but eventually needs to make a *Boolean* ("*yes or no*") *decision* before the raw data from the PADs being made accessible to the consumer as well as before derived data packed in PADs being allowed for output. To achieve this, we uses the same runtime infrastructure to allow policy engines be written in the same fashion as the consumer program. For a policy engine, it has to implement 2 policy check calls for the input and output pipelines. The input call will be called by the middleware before the consumer program tries to access the data. It takes the PADs in the dataset and the program as the input and returns a Boolean value for if the consumer program is allowed to access the data based on the ⟨input-constraints⟩ and ⟨program-constraints⟩ in the policies of these PADs. The output call will be called by the middleware when the consumer program requests an output. It takes the derived raw data, attributes and policy as well as the policies from the PADs in the dataset then returns a Boolean value for if the consumer program is allowed to output the derived data based on the ⟨output-constraints⟩. The corresponding pipeline will then allow or deny the input/output based on the returned Boolean status from the policy check call of the policy engine. An example policy engine design will be presented in § 5.

With this middleware, the burden of trust is now onto the middleware instead the consumer program. When the consumer program would like to read certain data, it must first gather all the PADs it needs to access and form the dataset. The dataset infrastructure will perform the decryption by contacting the key delegators of the PADs. Key delegators will perform a remote attestation to the middleware using hardware TEE's remote attestation mechanism to ensure that the middleware is genuine, and then provision the keys to the dataset infrastructure. After decrypting the PADs, the dataset infrastructure will run a policy check across all PADs using the policy engine. It will only provide raw data access to the consumer program if the policy check is passed on every PAD.

For data derivation, since policies may have output constraints, whenever the consumer program attempts to output any derived data, it must go through the output pipeline as illustrated in Figure 4. Any other output interfaces are disabled. The consumer program needs to act like a data producer to generate proposed data attributes and derived policy including custodian info for joint ownership. The data, attributes and policy will go through the policy engine to for compliance check against the input PADs' policies. Only if the output data, attributes and policy passes the check before they can be packed into a new PAD. Just like a producer, the new key will be sent to the new PAD's custodian and then the key delegators for further distribution.

4 Implementation

In this section, we provide details of our prototype implementation of Styx. We implemented a fully-functional system based on Intel

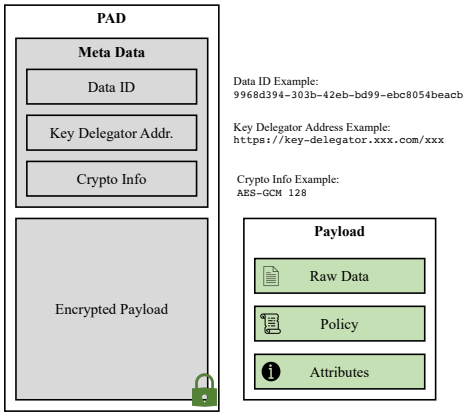

Figure 5: The format of PAD in our implementation. The left side shows the view of a PAD before decryption.

SGX [83] and WebAssembly [7] to demonstrate the feasibility and usability of our design. We have also successfully ported real-world applications into the system, including libonnx, an ONNX engine. The implementation consists of 13470 lines of code. Note that while we chose WebAssembly and SGX for this demo, the runtimes and TEEs are not limited to them and and can be swapped with other alternatives based on the need of the application. The source code of STYX is available at https://github.com/OSUSecLab/styx.

We follow our design philosophy that the data protocol and framework should be detached from a specific architecture so the data can be processed on heterogeneous processing nodes. Then on top of this, we build the demo system.

4.1 PAD Format

The PAD format essentially forms the data protocol that must be both architecture independent and secure. We followed the BNF presented in Figure 2 for the layout but added and specified a few fields in details. For all IDs used in PADs, we chose UUID to ensure that the ID is universally unique. We also added a "Crypto Info" field to represent the cryptographic algorithm used. This is to enable heterogeneous computation, particularly for smaller IoT devices with limited choices. To ensure both confidentiality and integrity, a cryptographic algorithm with CMAC can be used (e.g., AES-GCM).

4.2 Framework

We implemented our protocol into a framework library with configurable modules based on the role of the target program so that every role can use the same framework code base. There are 6 modules in the framework: *Data Packer*, *Dataset Management*, *Data Unpacker*, *Cryptographic Interface*, *Secret Management*, *Policy Engine Support*, *TEE Support*, *Runtime Support*. While different roles require different modules, a consumer middleware requires all of them and can better demonstrate the architecture on a single node. We therefore illustrate a consumer middleware architecture in Figure 6 and discuss each component in detail below.

Data Packer. The data packer provides interfaces for a program to pack data and policy into a PAD. It is needed by both producers and consumers. When the program wants to pack a piece of data

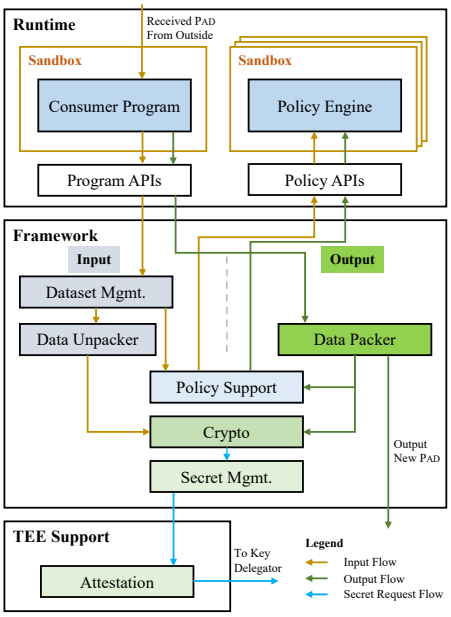

Figure 6: A consumer middleware hosting multiple policy engines and a consumer program.

with its policy, it provides the data packer with the elements inside the payload and the metadata. If the data packer is working in a consumer middleware, it will pass the raw data and proposed policy to the policy engine before packing the data. The packer will inquire the secret management to retrieve or generate the key, encrypt the payload and produce the PAD.

Dataset Management. The dataset management module is needed by the middleware of the consumer to build a one-shot dataset over which the policy is evaluated before granting the access to the consumer program. When the program wants to access a set of PADs, it will first have to create a dataset, add PADs into it and then request a policy check over the entire dataset before it can be allowed to access the data in the dataset.

Data Unpacker. The data unpacker is used by the dataset management module to extract data and policy from a PAD. When a PAD is added into the dataset, the data unpacker will inquire the secret management module to either retrieve the key locally or to fetch the secret key from the remote key delegator and then decrypt the payload. The plain-text payload is sent back to the dataset management module. Note that the program cannot access the plain-text payload until the policy check is passed.

Cryptographic Interface. To adapt to the diverse yet possibly limited cryptographic capabilities of different platforms, we designed a unified interface for cryptographic so that developers can easily add or configure the algorithms they want to support. This module is required on both producer and consumer. While optional, integrity check requires the algorithm to generate a CMAC.

Secret Management. The secret management module is one of the fundamental security gatekeepers of our design. It is required on all roles and serves as a mapping from the data ID to the data's key (the secret). The secret can be loaded locally or fetched from

the remote key delegator. The custodian of the data has the ability to push the data encryption key to the remote key delegator to make the key available to other programs. When the remote key delegator receives a fetch/push request, it will perform mutual attestations against the client connecting to it to prove its identity as well as ensuring the client is using an approved infrastructure (e.g., a genuine consumer middleware) on a trusted hardware.

Policy Engine Support. Each policy language has a policy engine. The engine is compiled into a software module that the runtime can execute. A policy engine module is loaded and identified by the policy type UUID. The policy evaluation interfaces are exposed to the dataset management and the data packer to check for compliance. We support multiple policy languages for different data by loading the corresponding policy engines.

TEE Support. In our design, all roles are protected within a TEE. Our framework can support different TEEs as long as the corresponding attestation and communication interfaces are implemented. The attestation must be mutual, meaning that when the attestation is completed, both sides can trust each other. The attestation should also exchange a ephemeral communication key so that the traffic between the two TEEs are encrypted. This channel is only necessary for secret management.

Runtime Support. The consumer program is sandboxed within a programming language runtime (e.g., WebAssembly, Python, etc.) to prevent it from leaking data unintentionally or deliberately. Styx also uses the same runtime to support policy engines. In our implementation, we specified a set of interfaces to manage the runtime and programs' lifecycle. To add support of a new runtime, a developer only needs to bind the runtime's interfaces to our interfaces for the framework to use the new runtime.

4.3 Integration with Intel SGX and WAMR

We built a demo prototype of Styx with Intel SGX as the TEE [65] and the WebAssembly Micro Runtime (WAMR) [7] as the runtime.

Integration with Intel SGX. Intel SGX provides an isolated and protected enclave environment within the user space memory. The enclave can access the data outside but the outside cannot access the data inside. This means that we can handle plain-text data inside the enclave without leaks. Intel SGX supports hardware-based remote attestation that verifies the integrity of the loaded software and the genuineness of the hardware. The attestation flow also creates an encrypted communication channel between the two enclaves using DH key exchange algorithm for the secret management module.

WAMR and Programs in the Middleware. WebAssembly (WASM) is a low-level IR that can be interpreted or even compiled natively into machine code. It can be sandboxed to its own memory space and its I/O is controlled by the runtime. Since WASM can be generated from LLVM IR, any programming language that LLVM supports can be compiled into WASM, allowing great flexibility for developers. WAMR is an embedded runtime that has full WASM support while being natively integrable into SGX.

To build a program using Styx, a developer can simply write a program that LLVM supports (e.g., C++) and call the interfaces we provided that are in pure C. In our design, for a program, we only export the dataset management interfaces to it and all the Pad accesses from the program must go through the dataset lifecycle by loading, checking policy and finally accessing. The I/O interfaces of the WAMR can be limited or even removed to prevent the program from leaking data without protection.

Similarly, to build a policy engine, a developer now implements the evaluation interfaces. The policy engine has full access to the plain-text data and policy to implement complex logics but has no I/O capability. It can also request the program's information to verify against the policy's constraints. The evaluation result is returned in a simple Boolean value indicating pass or fail.

5 Application Analysis

In this section, we present an analysis on how to apply our design to the motivating example discussed in §3.1, and how Styx can protect the training dataset and output model. Recall that in this use case, multiple hospitals provide confidential diagnostic data to jointly train a cancer classification model in a way that no party has direct access to the model parameters, and they can jointly approve policies to the access of the trained model.

We discuss the policy and consumer program design in details here, but we have implemented this scenario as well for feasibility and performance evaluation which we will present in § 6.

Training Data Preparation. Each hospital creates a Pad to protect its training diagnostic data. To do this, each hospital needs to use its own producer to generate the Pad and provision the data key received from the producer to a verified key delegator. Here, each hospital is the *custodian* of their own Pads. The key delegator can be offered by a cloud service provider as long as it is verifiable.

Jointly Training and Model Policy. The hospitals agree on a training program as the consumer program using the ⟨program-constraints⟩ that takes the input datasets in Pads and output the model protected by Pad as well. The data custodian of the output model is jointly controlled by the hospitals.

The policy attached to the training data, particularly the ⟨input-constraints⟩ can differ based on the considerations of each hospitals. For example, a hospital may require that their contributed training data shares no more than 20% of the total data; Another hospital may, in contrast, require that its share to be more than 50% so that it can have a higher quota of the resulting model. Only when all the policies are satisfied can the computation move forward.

The output Pad, which is the cancer classification model, will be attached with a policy generated by the consumer that is agreed by all the hospitals involved. It can be further limited by the ⟨output-constraints⟩ of each training data. For example, its policy can specify that the policy may be updated if agreed by entities who jointly control at least T of total share.

Using the Model. To query or fine tune the model, an organization is limited on the query quota based on a fair usage policy and the policy engine will keep a record of it. It must also use one of the agreed programs specified by the program constraint as the consumer program. The sandboxing mechanism ensures that the model can be used to answer queries without leaking. If the hospital wants to fine-tune the model to fit its own customized needs, it must use an approved fine-tuning program as the consumer program. An output ownership restriction policy can be specified to ensure that the

new model is restricted to this hospital, preventing the model being transferred to another entity without the agreement of all hospitals.

Example of Training Data Policy. To satisfy these requirements, an example policy of the training data can include the following:

- **Program Constraint.** The training data can only be used with programs approved by the hospital. The programs can be specified by, for example, hashes and encoded as a raw byte sequence.
- **Input Constraint.** The hospital can limit its contribution to not exceed certain percentage (e.g., 20%), as a regular integer.
- **Output Constraint.** The program's output, which is the model, must be under the control of the designated entity agreed by all the hospitals, which is a custodian represented by a UUID.

A detailed example of the policy enforcement is presented in our technical report in [84].

6 Performance Evaluation

We evaluated the performance of our complete Styx implementation described in § 4.3. We first evaluated the above example to show the overheads in the entire workflow of Styx and the scalability of Styx, then we provide a runtime benchmark overhead of the system. Our evaluations were conducted on a Dell PowerEdge R450 with an Intel Xeon Silver 4314 CPU running at 2.4 GHz and 128 GiB of memory among which 2 GiB can be used as the enclave memory.

To evaluate our system, we first implemented the scenario of hospitals jointly training a cancer classification model discussed in § 5 using the Diagnostic Wisconsin Breast Cancer Database [76] to train an SVM model with libsvm [13]. We use it as one of our subjects and analysed the entire workflow, from input to output, to discuss the overheads of each step in § 6.1. We used the broken down performance figures with a simulation technique to evaluate the scalability of Styx on a distributed setting. To demonstrate the capability of handling complex applications, we also ported libonnx [37] and NBench [47] to demonstrate the overall performance when a runtime is involved in § 6.3.

6.1 ML Training Workflow Evaluation

We used the Diagnostic Wisconsin Breast Cancer Database [76] to train an SVM model using libsvm [13] for the workflow overheads evaluation. To provide more insights on the performance of input of Styx, we set 3 hospitals (A, B and C) involving in the training, each provides 3 copies of the database for a total of 9 pieces of data coming from 3 different entities. For each piece of data, the policy discussed in § 5 is attached with proper attributes. The output model is restricted by an ⟨output-constraint⟩ in the policy that it must be under the custody of hospital A.

We implemented the policy engine according to the policy language design. The timeline breakdown of the training process is illustrated in Figure 7. One can easily see that there are 3 events that claimed the most overheads in the workflow: Load B-1, Load C-1 and Train Model. Since the consumer program is owned by hospital A, when first loading dataset of hospital B and C, it will need to contact the key delegator to fetch the data keys which involves communications over sockets and remote attestations. However, once the key is fetched to local, when loading the data from the same entity, Styx no longer needs to communicate with the key delegator and therefore will be much faster.

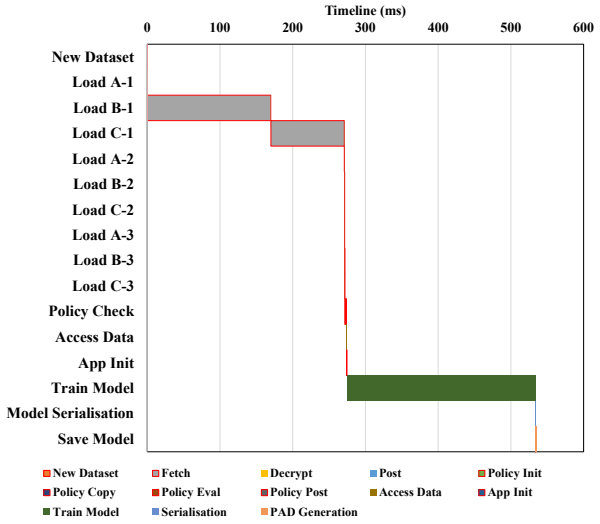

Figure 7: Timeline breakdown of the jointly training example using Styx. Red outline means the overhead is not related to the data's size.

Table 1: Average overheads breakdown of the framework of Styx measured in ms. Gray overhead title means the overhead is not related to the data's size.

		w/ Fetching	w/o Fetching
New Dataset			0.007
Load	Attestation	135.089	-
	Fetch Key	0.177	0.007
	Decrypt	0.088	0.098
	Policy Matching	0.007	0.008
	Total	135.362	0.113
Policy Eval	Policy Init		1.544
	Policy Copy		0.138
	Policy Eval		0.366
	Policy Post		0.025
	Total		2.073
Access Data			0.258
Output	Policy Init		0.119
	Policy Copy		0.074
	Policy Eval		0.330
	Policy Post		0.009
	Generate Pad		1.405
	Total		1.937

The exact value of the overheads are provided in Table 1. We can see that most of the overheads are below 2 ms. When fetching the key with the remote attestation and communication involved, we can see that the overheads were around 135 ms, which is comparable to common encrypted transmission methods like HTTPS [54].

Note that a red outline in Figure 7 or a gray title in Table 1 indicates that the overhead is not related to the data's size, meaning that no matter how large the data is, it will always cost the same. We can see that all the overheads that scale with the data's size were lower than 0.4 ms, meaning that Styx can maintain a good scalability and low overhead even with heavy inputs and outputs.

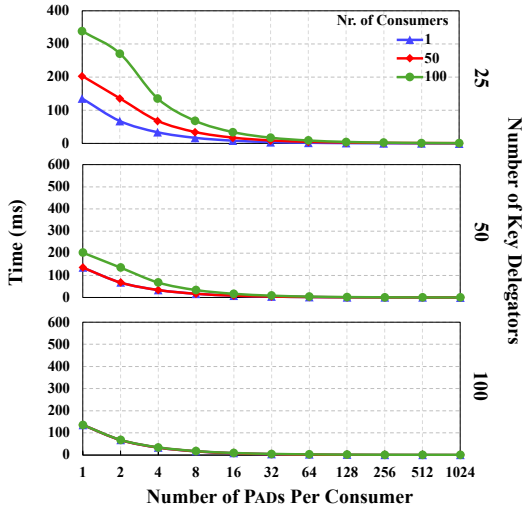

Figure 8: Per-PAD latency on large scale distributed deployment from simulation.

It is worth noting that the overheads of the policy check significantly depend on how complex the logic of the policy engine implements. As a module for the runtime, the performance is also highly related to the runtime's to be discussed in § 6.3. For the input and output constraints, the scalability of the policy check is linked to the size of the dataset as each PAD's policy has to be evaluated. For the program constraints, the overhead is static to a specific consumer program and does not increase with the dataset.

6.2 Scalability on Distributed Deployment

As a distributed design, STYX is expected to perform stably at large scale. To evaluate the scalability characteristics of STYX, we employed a similar methodology from [64] to simulate the performance of STYX at heavy load with a large amount of nodes. We consider the key delegators behind a simple load balancer that dispatches a key fetching request to whichever key delegator that is free. Once a key delegator has attested a consumer, the upcoming key fetching no longer needs attestation.We sampled the latency of attestations and key requests with expectation values and standard deviations obtained from our experiments on real machine. In our settings, we simulate a number of consumers trying to decrypt certain amount of PADs all at once to present a heavy load scenario. We simulated with 3 different numbers of key delegators and consumers, with each consumer trying to access different amount of PADs ranging from 1 to 1024.The results are illustrated in Figure 8.

From the figure, we can easily observe the trend that the more PADs a consumer needs to decrypt, the lower the latency it eventually needs. This is because the heavy attestation only has to be done once, and the subsequent requests only need the key fetching. When the amount of consumers is smaller than the key delegators, the per-PAD latency is capped at a single attestation plus the key fetching latency. This is because every consumer can have a dedicated key delegator serving it. When the amount of consumers is larger than the key delegators, the max latency is in a linear scale to the amount of consumers minus the amount of delegators. From the overall results, we can see that even under extreme heavy loads with hundreds of consumers and thousands of PADs to be decrypted, STYX can still achieve a per-PAD latency at max to a couple hundreds of milliseconds.

6.3 Runtime Performance and Considerations on Complex Applications

Since STYX involves runtime sandboxing and TEE environments, we present two computation-intensive applications: NBench [47] and libonnx [37] to demonstrate the performance overheads of STYX when using the WAMR runtime. The difference between the two benchmarks is that for NBench, we compiled it into WASM AoT binaries to demonstrate the slow down when heavy computation is done inside the runtime; For libonnx, we ported it as a trusted native binary that the WASM code can call for computation.

Runtime Computation Performance with NBench. For the vanilla NBench, it was compiled using the GCC with an optimization level of O3. For the WebAssembly version, it was first compiled using WASI-SDK 20.0 [12], then compiled to AoT binaries, all with an optimization level of O3. The result is normalized to the vanilla's score, that is, how many times slower is presented in Figure 9.

As we can see that WAMR imposed slowdown on all benchmarks. While most benchmarks experienced a slowdown around 1.5x, STRING SORT and NEURAL NET suffered the most. However, the geometric mean of the slowdown is around 2.2x which is also close to the benchmark result presented in [20] (~2.3x).

Note that WAMR, as chosen in our system implementation, is just one option for the runtime. Other runtimes can also be used to better suit the characteristics of the workload. Alternatives like Python [45] and JavaScript (using NodeJS with V8 engine) [44] showed different levels of slowdown ranged from 2x as the WASM or up to 20x. A runtime provides the sandboxing required in STYX as well as offers better flexibility and safety for developers. We believe that this is a worthy trade-off over pure C code.

Native Porting with libonnx. By porting a trusted computation-heavy library natively, we can achieve an almost identical performance to native computation. To demonstrate it as well as presenting the capability of handling complex tasks of STYX, we chose the libonnx as our subject. libonnx is an execution engine of ONNX [56], a language for encoding ML models. This means that a jointly-trained model can also be encoded into ONNX format.

We linked libonnx into the middleware as a trusted native library and tested the performance with libonnx's benchmark suite as the consumer program. The models were packed into PADs with the policies as specified in Figure 1. We compared STYX with a vanilla native libonnx on Linux and normalized the benchmark show the slowdown in Figure 10. For the total time of the 6 benchmark, the slowdowns are ranging from 0.9x to 1.1x. We can even observe speed-ups in `mnist_8` and `super_resolution_10`. We suspect this is due to the difference of the memory allocation behavior between the Intel SGX versus a regular Linux. In Intel SGX, heap is pre-allocated so the allocator does not have use syscalls.

For certain sub-benchmarks showing "significant" slowdowns over 3x, the slowdowns are due to the fact that these sub-benchmarks were so fast that their execution time was way smaller than the WASM calling overhead (around 3 us). This made the

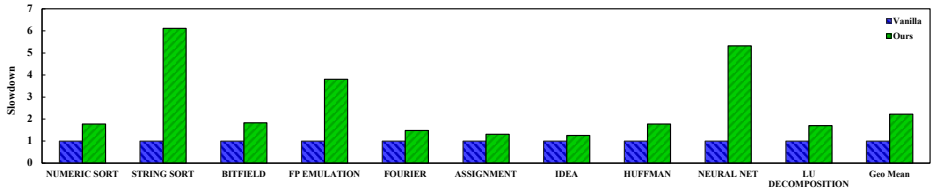

Figure 9: Normalized NBench slowdown of WAMR over a vanilla result.

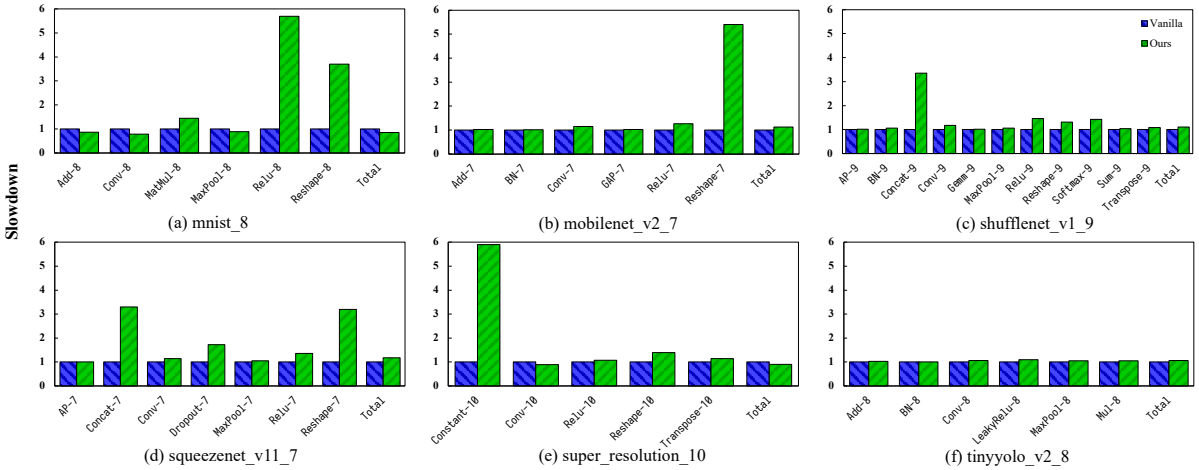

Figure 10: Normalized libonnx benchmark slowdown of the 6 models provided.

slowdown looks huge but was not actually impacting the overall performance. Also, the geometric mean of the slowdowns of the 6 benchmarks is 1.03x, making our system on par with a native one when handling tasks using trusted libraries like libonnx.

Porting Consideration. When comparing the two benchmarks, we can see that porting a trusted library natively can greatly help minimizing the overheads. This is suitable for cases where open source libraries are involved. This may also be required when certain external hardware or system features are used that cannot be directly accessed within WASM. However, even with pure WASM code, the overheads were still manageable, allowing more flexibility for developers to implement their propriety code with Styx.

7 Discussion

7.1 TCB Scope and Alternative Isolation

TCB Scope. From our threat model discussed in § 3.3 and the architecture introduced in § 3, we can see that on top of the basic supportive software of the TEE, we added the Styx framework, the runtime and the policy engines into the TCB. For our SGX-based prototype, the basic supportive software is the libraries of SGX SDK.

CVM-Based TEEs. While our prototype is enclave-based, our design similarly works with confidential VMs (CVMs). Instead of protecting a small piece of software, these CVMs isolates on a VM level granularity. In a similar fashion, we can run the software stack as a process inside the CVM. However, now the supportive software stack will need to include a small OS as a host. A lightweight OS can be sufficient for the purpose such as the one proposed in Gramine-TDX [43]. When compared to an enclave-based solution, CVM-based TEEs will inevitably increase the size of the TCB.

Process Isolation. TEEs are employed to help defend threats from the outside, particularly against privileged attackers such as the computation infrastructure provider. However, if the infrastructure provider can be trusted, then the only threat will be from the consumer program. In this case, process isolation can be sufficient by running Styx as a process given that the OS it runs upon is trusted.

7.2 Limitations and Future Works

While Styx introduces a novel approach to secure data processing in collaborative environments, our current implementation and design are subject to certain limitations. These areas, however, also offer opportunities for enhancements and further works.

Policy Enforcement in Hardware Accelerators. Currently, Styx lacks the capability to enforce policies within hardware accelerators, such as GPUs. This gap is notable as TEE support in GPUs has shown up recently [55]. Extending Styx to integrate with these new TEEs in hardware accelerators represents an important direction for future work, potentially broadening the applicability of Styx to more tailored computing scenarios, particularly those in AI.

Security-Performance Balance in Runtime Sandboxing. Styx faces a potential trade-off between security and performance, particularly concerning runtime sandboxing. The practice of compiling bytecode to native machine code ahead-of-time (AoT) for performance improvements may compromise the sandboxing mechanism, as the behaviors of AoT binaries might not be thoroughly validated. However, methods like Software Fault Isolation (SFI) and containerization, as discussed in [85] and [5], provide promising strategies to ensure robust sandboxing even for AoT binaries. The integration of such techniques is left for future investigation.

Middleware Verification. The consumer middleware of the Styx framework may benefit from formal verification, which can improve the robustness to the system's overall security [14, 19, 36, 38, 74]. While formal verification imposes a substantial amount of work, precedents set by projects like [68] has demonstrated the feasibility and value of such efforts. Additionally, the application of SFI, akin to those mentioned in [85], could offer interim safeguards for the runtime, even in the absence of formal verification.

8 Related Works

8.1 Sticky Policy Enforcement

Pad or *sticky policy* is used in Styx to embed policies into data. While recent surveys [48, 60] showed advancement on the sticky policy's design, the enforcement of such policies is the key to the security guarantee. Existing works focus on encryption based techniques. Examples such as Identity-Based Encryption [10], Attribute-Based Encryption [8, 30], and Proxy Re-encryption [31] fall into this category. These encryption-based enforcement requires a data consumer to have proper keys that represents the identity or privilege for the access. If all the necessary keys are met under the policy proving the permission, the keys will be able to decrypt the data based on the cryptographic design.

Compared with Styx, existing works have limitations on the expressiveness of the policy. They also lack the support to impose policies on applications and derived outputs. This means that once the policy check is passed and the data is decrypted, the usage of the data and the outputs will not be in the control of the data owner and therefore cannot satisfy the data-in-use protection we envisioned.

8.2 Multi-Stakeholder Computing and ML

There have been efforts on securing multi-stakeholder computing where stakeholders are mutually distrusted. However, most of the existing works focus on computation configurations and rely heavily on the perfection of the code via verification. For example, Palæmon offers a computation policy enforcement with a combination of human effort certification and TEE reporting [32]. R. Walther, et al. presented a computation configuration policy enforcement for untrusted cloud [73]. Compared with Styx, the two methods focused on the code of the data consumers and cannot provide flexible per-data policy enforcement throughout the data lifecycle. They also did not address the dynamic collaboration problem. S. Tokuda, et al. proposed a static taint analysis method to enable arbitrary code loading while maintaining certain properties of the data security [70]. This, however, relies on assumptions that the consumer program must be simple enough for the static analysis before loading, limiting what the consumer program can do, not to mention the lack of data lifecycle protection and dynamic collaboration.

Given our motivating example, multi-stakeholder ML is also related to Styx. Confidential Federated Learning (CFL) is one way to achieve almost full control of the sensitive data locally [33]. However, CFL lacks protection on the output model, and cannot achieve policies such as percentage limit of one entity in the entire training dataset. However, we believe that Styx can be used in combination with existing methods like CFL to achieve such control.

8.3 Middleware Sandboxing

The dominant technique for data-in-use protection of Styx is middleware sandboxing. Existing works on middleware sandboxing have explored more on simple access controls with either limited policy and data-in-use protection or limited use cases.

Twine [52] offers a WASM sandbox that can host trusted workloads with transparent encrypted I/O. This works in an opposite way comparing to our method where the workload itself is actually untrusted and the I/O of which is strictly limited.

Ryoan [35] achieved sandboxing via NaCL [80] and controls the I/O via ownership labeling. Data being processed on an entity will have the entity's ownership label added. An entity can remove its label once it has finished the processing and make sure that there is no sensitive data from it is left. Compared with Styx, Ryoan only offers control on ownership level with no possibility to specify complex policies like the inter-Pad rules Styx supports. It also has no policy control on the application and the derived data, meaning that once the data is processed and new data is created, the new data is completely out of the original data owner's control.

Laputa [40] offers private data analysis on a TEE with policy check. It takes a specialized form of code called query plans, and performs policy check with policies defined using regular expressions. Compared with Styx, Laputa cannot execute generic code and can only take the special form of query plan. It also does not take multiple pieces of data like Styx for a collaborative scenario. The policy, while flexible, can only be applicable to the query plan form. There is also no policy control on the derived output, which is expected to contain no sensitive information via the policy check.

When compared with Styx, existing works on middleware sandboxing focus on the input side of a single piece of data and control under what circumstances the data can be read. They, however, do not handle derived policies for derived data, and have limited support on the input and application policies for multiple pieces of data from different stake holders under a collaborative setting.

9 Conclusions

We have presented Styx, a novel framework that combines the flexibility of sticky policy and strong enforcement of TEEs to ensure data protection throughout its lifecycle in distributed collaborative settings. Styx achieved data-in-use protection, data lifecycle protection and dynamic collaboration via runtime sandboxing using a middleware. Our application analysis and system implementation demonstrated the feasibility, effectiveness and scalability while imposing reasonable overheads similar to conventional encrypted communication channels even under a distributed multi-node deployment. With this design, Styx can enforce per-data policy from multiple stakeholders throughout the entire data lifecycle for modern collaborative workloads like AI training and other distributed privacy-concerning data processing scenarios.

Acknowledgments

We would like to thank the reviewers and shepherd for their feedback and suggestions. The authors from The Ohio State University were partially supported by NSF award 2207202. The authors from Purdue University were partially supported by NSF award 2207204.

References

[1] Amazon Web Services, Inc. 2011. AWS Identity and Access Management (IAM). https://aws.amazon.com/iam/.

[2] Amazon Web Services, Inc. 2020. AWS Key Management Service. https://aws.amazon.com/kms/.

[3] Amazon Web Services, Inc. 2020. AWS Nitro Enclaves. https://aws.amazon.com/ec2/nitro/nitro-enclaves/.

[4] AMD. 2020. Strengthening VM Isolation with Integrity Protection and More. *White Paper, January* 53 (2020), 1450–1465.

[5] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 689–703. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov

[6] N Asokan, Jan-Erik Ekberg, Kari Kostiainen, Anand Rajan, Carlos Rozas, Ahmad-Reza Sadeghi, Steffen Schulz, and Christian Wachsmann. 2014. Mobile Trusted Computing. *Proc. IEEE* 102, 8 (2014), 1189–1206.

[7] Web Assembly. [n. d.]. WebAssembly. https://webassembly.org. (Accessed on 30/04/2022).

[8] John Bethencourt, Amit Sahai, and Brent Waters. 2007. Ciphertext-Policy Attribute-Based Encryption. In *2007 IEEE Symposium on Security and Privacy (SP'07)*. IEEE, 321–334.

[9] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, et al. 2009. Secure Multiparty Computation Goes Live. In *Financial Cryptography and Data Security: 13th International Conference, FC 2009, Accra Beach, Barbados, February 23-26, 2009. Revised Selected Papers 13*. Springer, 325–343.

[10] Dan Boneh and Matt Franklin. 2001. Identity-Based Encryption from the Weil Pairing. In *Annual International Cryptology Conference*. Springer, 213–229.

[11] Zvika Brakerski and Vinod Vaikuntanathan. 2011. Efficient Fully Homomorphic Encryption from (Standard) LWE. *SIAM J. Comput.* 43, 2 (2011), 831–871. https://eccc.weizmann.ac.il/report/2011/078/

[12] Andrew Brown. [n. d.]. WebAssembly/wasi-sdk: WASI-enabled WebAssembly C/C++ toolchain. https://github.com/WebAssembly/wasi-sdk. (Accessed on 28/11/2023).

[13] Chih-Chung Chang and Chih-Jen Lin. [n. d.]. cjlin1/libsvm: LIBSVM – A Library for Support Vector Machines. https://github.com/cjlin1/libsvm (Accessed on 14/04/2024).

[14] Fabrizio Cicala, Weicheng Wang, Tianhao Wang, Ninghui Li, Elisa Bertino, Faming Liang, and Yang Yang. 2021. Pure: A Framework for Analyzing Proximity-Based Contact Tracing Protocols. *ACM Computing Surveys (CSUR)* 55, 1 (2021), 1–36.

[15] Google Cloud. 2020. Google Cloud Key Management Service. https://cloud.google.com/kms/.

[16] Google Cloud. 2020. Google Cloud Shielded VMs. https://cloud.google.com/security/shielded-cloud/shielded-vm.

[17] Luigi Coppolino, Salvatore D'Antonio, Giovanni Mazzeo, and Luigi Romano. 2025. An Experimental Evaluation of TEE Technology: Benchmarking Transparent Approaches Based on SGX, SEV, and TDX. *Computers & Security* 154 (2025), 104457. doi:10.1016/j.cose.2025.104457

[18] Ronald Cramer, Ivan Bjerre Damgård, et al. 2015. *Secure Multiparty Computation*. Cambridge University Press.

[19] Haotian Deng, Weicheng Wang, and Chunyi Peng. 2018. Ceive: Combating Caller ID Spoofing on 4G Mobile Phones via Callee-Only Inference and Verification. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*. 369–384.

[20] Frank Denis. [n. d.]. Performance of WebAssembly runtimes in 2023 | Frank DENIS random thoughts. https://00f.net/2023/01/04/webassembly-benchmark-2023/. (Accessed on 28/11/2023).

[21] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2016. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In *International Conference on Machine Learning (ICML)*. http://proceedings.mlr.press/v48/dowlin16.pdf

[22] Zvi Galil, Stuart Haber, and Moti Yung. 1987. Cryptographic Computation: Secure Fault-Tolerant Protocols and the Public-Key Model. In *Conference on the Theory and Application of Cryptographic Techniques*. Springer, 135–155.

[23] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. 2003. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*. 193–206.

[24] Craig Gentry. 2009. *A Fully Homomorphic Encryption Scheme*. Technical Report STAN-CS-09-XXX. Stanford University. https://crypto.stanford.edu/craig/craig-thesis.pdf

[25] Craig Gentry and Shai Halevi. 2011. Implementing Gentry's Fully-Homomorphic Encryption Scheme. In *Proceedings of the 31st Annual International Conference on Advances in Cryptology (CRYPTO '11)*. 129–148. https://eprint.iacr.org/2011/133.pdf

[26] Dimitra Giantsidi, Julian Pritzi, Felix Gust, Antonios Katsarakis, Atsushi Koshiba, and Pramod Bhatotia. 2025. TNIC: A Trusted NIC Architecture: A Hardware-Network Substrate for Building High-Performance Trustworthy Distributed Systems. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 1282–1301. doi:10.1145/3676641.3716277

[27] Alexanda S Gillis. 2019. Confidential Computing. https://www.techtarget.com/searchcloudcomputing/definition/confidential-computing

[28] Oded Goldreich, Silvio Micali, and Avi Wigderson. 2019. How to Play Any Mental Game, or a Completeness Theorem for Protocols with Honest Majority. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*. 307–328.

[29] Google. [n. d.]. View Build Provenance | Cloud Build Documentation | Google Cloud. https://cloud.google.com/build/docs/securing-builds/view-build-provenance

[30] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. 2006. Attribute-Based Encryption for Fine-Grained Access Control of Encrypted Data. In *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, October 30 - November 3, 2006*, Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati (Eds.). ACM, 89–98. doi:10.1145/1180405.1180418

[31] Matthew Green and Giuseppe Ateniese. 2007. Identity-Based Proxy Re-Encryption. In *Applied Cryptography and Network Security: 5th International Conference, ACNS 2007, Zhuhai, China, June 5-8, 2007. Proceedings 5*. Springer, 288–306.

[32] Franz Gregor, Wojciech Ozga, Sébastien Vaucher, Rafael Pires, Do Le Quoc, Sergei Arnautov, André Martin, Valerio Schiavoni, Pascal Felber, and Christof Fetzer. 2020. Trust Management as a Service: Enabling Trusted Execution in the Face of Byzantine Stakeholders. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 502–514. doi:10.1109/DSN48063.2020.00063

[33] Jinnan Guo, Peter Pietzuch, Andrew Paverd, and Kapil Vaswani. 2024. Trustworthy AI Using Confidential Federated Learning. *Commun. ACM* 67, 9 (Aug. 2024), 48–53. doi:10.1145/3677390

[34] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. 2011. Faster Secure Two-Party Computation Using Garbled Circuits. In *20th USENIX Security Symposium (USENIX Security 11)*.

[35] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. 2016. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 533–549. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/hunt

[36] Syed Hussain, Omar Chowdhury, Shagufta Mehnaz, and Elisa Bertino. 2018. LTEInspector: A Systematic Approach for Adversarial Testing of 4G LTE. In *Network and Distributed Systems Security (NDSS) Symposium 2018*.

[37] Jianjun Jiang. [n. d.]. xboot/libonnx: A lightweight, portable pure C99 onnx inference engine for embedded devices with hardware acceleration support. https://github.com/xboot/libonnx (Accessed on 14/04/2024).

[38] Imtiaz Karim, Abdullah Al Ishtiaq, Syed Rafiul Hussain, and Elisa Bertino. 2023. BLEDiff: Scalable and Property-Agnostic Noncompliance Checking for BLE Implementations. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 3209–3227.

[39] Günter Karjoth, Matthias Schunter, and Michael Waidner. 2002. Platform for Enterprise Privacy Practices: Privacy-Enabled Management of Customer Data. In *Privacy Enhancing Technologies, Second International Workshop, PET 2002, San Francisco, CA, USA, April 14-15, 2002, Revised Papers (Lecture Notes in Computer Science, Vol. 2482)*. Springer, 69–84.

[40] Byeongwook Kim, Jaewon Hur, Adil Ahmad, and Byoungyoung Lee. 2025. Secure Data Analytics in Apache Spark with Fine-grained Policy Enforcement and Isolated Execution. In *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/secure-data-analytics-in-apache-spark-with-fine-grained-policy-enforcement-and-isolated-execution/

[41] Steven L Kinney. 2006. *Trusted Platform Module Basics: Using TPM in Embedded Systems*. Elsevier.

[42] 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 40th IEEE Symposium on Security and Privacy, SP 2019*. IEEE.

[43] Dmitrii Kuvaiskii, Dimitrios Stavrakakis, Kailun Qin, Cedric Xing, Pramod Bhatotia, and Mona Vij. 2024. Gramine-TDX: A Lightweight OS Kernel for Confidential VMs. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 4598–4612. doi:10.1145/3658644.3690323

[44] Programming Language and compiler Benchmarks. 2024. C VS Javascript benchmarks. https://programming-language-benchmarks.vercel.app/c-vs-javascript. (Accessed on 26/01/2024).

[45] Programming Language and compiler Benchmarks. 2024. C VS Python benchmarks. https://programming-language-benchmarks.vercel.app/c-vs-python. (Accessed on 26/01/2024).

[46] Dahlia Malkhi, Noam Nisan, Benny Pinkas, Yaron Sella, et al. 2004. Fairplay-Secure Two-Party Computation System.. In *USENIX security symposium*, Vol. 4. San Diego, CA, USA, 9.

[47] Uwe F. Mayer. [n. d.]. Linux/Unix nbench. https://www.math.utah.edu/~mayer/linux/bmark.html. (Accessed on 28/11/2023).

[48] Daniele Miorandi, Alessandra Rizzardi, Sabrina Sicari, and Alberto Coen-Porisini. 2020. Sticky Policies: A Survey. *IEEE Transactions on Knowledge and Data Engineering* 32, 12 (2020), 2481–2499. doi:10.1109/TKDE.2019.2936353

[49] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 19–38.

[50] Marco Casassa Mont, Siani Pearson, and Pete Bramhall. 2003. Towards Accountable Management of Identity and Privacy: Sticky Policies and Enforceable Tracing Services. In *14th International Workshop on Database and Expert Systems Applications (DEXA'03), September 1-5, 2003, Prague, Czech Republic*. IEEE Computer Society, 377–382. doi:10.1109/DEXA.2003.1232051

[51] Steven J Murdoch. 2015. Introduction to Trusted Execution Environments (TEE)—IY5606. *CiteSeerx: University Park, PA, USA* (2015).

[52] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, Valerio Schiavoni, Giovanni Mazzeo, Arne Hollum, and Darshan Vaydia. 2024. A Comprehensive Trusted Runtime for WebAssembly With Intel SGX. *IEEE Transactions on Dependable and Secure Computing* 21, 4 (2024), 3562–3579. doi:10.1109/TDSC.2023.3334516

[53] Nataraj Nagaratnam. 2020. What is confidential computing? https://www.ibm.com/cloud/learn/confidential-computing

[54] David Naylor, Alessandro Finamore, Ilias Leontiadis, Yan Grunenberger, Marco Mellia, Maurizio Munafò, Konstantina Papagiannaki, and Peter Steenkiste. 2014. The Cost of the "S" in HTTPS. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies* (Sydney, Australia) *(CoNEXT '14)*. Association for Computing Machinery, New York, NY, USA, 133–140. doi:10.1145/2674005.2674991

[55] NVDIA. [n. d.]. Confidential Computing | NVIDIA. https://www.nvidia.com/en-us/data-center/solutions/confidential-computing/. (Accessed on 27/1/2024).

[56] ONNX. [n. d.]. ONNX | Home. https://onnx.ai. (Accessed on 25/01/2024).

[57] Julian Padget and Wamberto W Vasconcelos. 2015. Policy-Carrying Data: A Step Towards Transparent Data Sharing. *Procedia Computer Science* 52 (2015), 59–66.

[58] Jaehong Park and Ravi Sandhu. 2002. Towards usage control models: beyond traditional access control. In *Proceedings of the seventh ACM symposium on Access control models and technologies*. 57–64.

[59] Jaehong Park and Ravi S. Sandhu. 2004. The UCON_{ABC} Usage Control Model. *ACM Transaction on Information System Security* 7, 1 (2004), 128–174. doi:10.1145/984334.984339

[60] Siani Pearson and Marco Casassa Mont. 2011. Sticky Policies: An Approach for Managing Privacy across Multiple Parties. *Computer* 44, 9 (2011), 60–68. doi:10.1109/MC.2011.225

[61] Ron Perez. 2021. Case study: Confidential Computing. https://www.intel.es/content/dam/www/central-libraries/us/en/documents/confidential-computing-case-studies.pdf

[62] Matthieu Pigaglio, Joachim Bruneau-Queyreix, Yérom-David Bromberg, Davide Frey, Etienne Rivière, and Laurent Réveillère. 2022. RAPTEE: Leveraging Trusted Execution Environments for Byzantine-Tolerant Peer Sampling Services. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. 603–613. doi:10.1109/ICDCS54860.2022.00064

[63] Alexander Pretschner, Manuel Hilty, and David A. Basin. 2006. Distributed Usage Control. *Commun. ACM* 49, 9 (2006), 39–44. doi:10.1145/1151030.1151053

[64] Patrick Sabanic, Masanori Misono, Teofil Bodea, Julian Pritzi, Michael Hackl, Dimitrios Stavrakakis, and Pramod Bhatotia. 2025. Confidential Serverless Computing. arXiv:2504.21518 [cs.CR] https://arxiv.org/abs/2504.21518

[65] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. 2017. Panoply: Low-TCB Linux Applications With SGX Enclaves.. In *NDSS*.

[66] Arm Blueprint staff. 2021. What Is Confidential Computing? Here's A Great Example. https://www.arm.com/blogs/blueprint/confidential-computing

[67] François-Xavier Standaert. 2010. Introduction to Side-Channel Attacks. *Secure integrated circuits and systems* (2010), 27–42.

[68] Mingshen Sun and Ram Rachum. [n. d.]. mesalock-linux/mesapy: A Fast and Safe Python based on PyPy. https://github.com/mesalock-linux/mesapy. (Accessed on 27/1/2024).

[69] Microsoft team. 2022. Azure confidential computing: Use cases and scenarios. https://learn.microsoft.com/en-us/azure/confidential-computing/use-cases-scenarios

[70] Shota Tokuda, Shohei Kakei, Yoshiaki Shiraishi, and Shoichi Saito. 2025. Decentralized Data Usage Control with Confidential Data Processing on Trusted Execution Environment and Distributed Ledger Technology. In *Network and System Security*, Houbing Herbert Song, Roberto Di Pietro, Saed Alrabaee, Mohammad Tubishat, Mousa Al-kfairy, and Omar Alfandi (Eds.). Springer Nature Singapore, Singapore, 127–144.

[71] Vinod Vaikuntanathan. 2010. Computing Blindfolded: New Developments in Fully Homomorphic Encryption. *Bulletin of the EATCS* 100 (2010), 31–54. https://eccc.weizmann.ac.il/report/2010/131/

[72] Amit Vasudevan, Emmanuel Owusu, Zongwei Zhou, James Newsome, and Jonathan M McCune. 2012. Trustworthy Execution on Mobile Devices: What Security Properties Can My Mobile Platform Give Me?. In *Trust and Trustworthy Computing: 5th International Conference, TRUST 2012, Vienna, Austria, June 13-15, 2012. Proceedings 5*. Springer, 159–178.

[73] Robert Walther, Carsten Weinhold, Peter Amthor, and Michael Roitzsch. 2024. Multi-Stakeholder Policy Enforcement for Distributed Systems. In *Proceedings of the 10th International Workshop on Container Technologies and Container Clouds* (Hong Kong, Hong Kong) *(WoC '24)*. Association for Computing Machinery, New York, NY, USA, 7–12. doi:10.1145/3702637.3702958

[74] Weicheng Wang, Fabrizio Cicala, Syed Rafiul Hussain, Elisa Bertino, and Ninghui Li. 2020. Analyzing the Attack Landscape of Zigbee-Enabled IoT Systems and Reinstating Users' Privacy. In *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. 133–143.

[75] Weicheng Wang, Hyunwoo Lee, Yan Huang, Elisa Bertino, and Ninghui Li. 2023. Towards Efficient Privacy-Preserving Deep Packet Inspection. In *European Symposium on Research in Computer Security*. Springer, 166–192.

[76] Wolberg William, Mangasarian Olvi, Street Nick, and Street W. [n. d.]. Breast Cancer Wisconsin (Diagnostic). https://archive.ics.uci.edu/dataset/17/breast+cancer+wisconsin+diagnostic (Accessed on 14/04/2024).

[77] Shouhuai Xu and Ravi Sandhu. 2004. Applying OM-AM to Analyze Digital Rights Management. In *7th International Conference on E-Commerce Research*, Vol. 53.

[78] Andrew C. Yao. 1982. Protocols for Secure Computations. In *Proceedings of the 23rd Annual IEEE Symposium on Foundations of Computer Science*.

[79] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets. In *Annual Symposium on Foundations of Computer Science*.

[80] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *2009 30th IEEE Symposium on Security and Privacy, SP 2009*. IEEE.

[81] Xinwen Zhang, Francesco Parisi-Presicce, Ravi S. Sandhu, and Jaehong Park. 2005. Formal Model and Policy Specification of Usage Control. *ACM Transactions on Information System Security* 8, 4 (2005), 351–387. doi:10.1145/1108906.1108908

[82] Xinwen Zhang, Jean-Pierre Seifert, and Ravi Sandhu. 2008. Security Enforcement Model for Distributed Usage Control. In *2008 IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC 2008)*. IEEE, 10–18.

[83] Shixuan Zhao, Mengyuan Li, Yinqian Zhangyz, and Zhiqiang Lin. 2022. vSGX: Virtualizing SGX Enclaves on AMD SEV. In *2022 IEEE Symposium on Security and Privacy (SP)*. 321–336. doi:10.1109/SP46214.2022.9833694

[84] Shixuan Zhao, Weicheng Wang, Ninghui Li, and Zhiqiang Lin. 2026. Styx: Collaborative and Private Data Processing With TEE-Enforced Sticky Policy. arXiv:2604.04082 [cs.CR] https://arxiv.org/abs/2604.04082

[85] Shixuan Zhao, Pinshen Xu, Guoxing Chen, Mengya Zhang, Yinqian Zhang, and Zhiqiang Lin. 2023. Reusable Enclaves for Confidential Serverless Computing. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 4015–4032. https://www.usenix.org/conference/usenixsecurity23/presentation/zhao-shixuan