Glasswall: Enforcing User-Defined Data Usage Policies in Functions as a Service

Silvan Niederer[0009−0009−2142−7728], Ali Hajiabadi[0000−0002−3219−7544], and Kaveh Razavi[0000−0002−8588−7100]

ETH Zurich, Switzerland

Abstract. Current data privacy regulations demand user consent to determine the legitimacy of data processing. Unfortunately, this requirement is open to interpretation and difficult to enforce in practice. This paper introduces Glasswall, a new system that augments existing Functions-as-a-Service (FaaS) platforms to enforce user-defined data usage policies by addressing three challenges. First, to provide end-to-end data protection, Glasswall does not trust either cloud or function providers, and instead builds on a trusted Key Distribution Server (KDS). Second, to preserve compatibility with existing functions while enforcing user-defined data usage policies, Glasswall features an information flow control mechanism with support for *permanent* declassification when requested by the user (e.g., file sharing) and *transient* declassification when handling user secrets (e.g., password checks). Third, to ensure that functions do not leak secrets across requests, Glasswall implements efficient process-level function snapshots that are resumed *per request*. The Glasswall reference implementation is the first solution compatible with existing FaaS platforms that protects user data without relying on either the cloud or function providers or indiscriminately restricting network capabilities. Our evaluation shows that Glasswall introduces overheads similar to prior work (4%-62% depending on the workload) while providing stronger security guarantees and preserving compatibility.

1 Introduction

Legislation such as the GDPR aim to protect user data from unauthorized access. While service providers are now obliged to inform users and require consent about how their data is going to be used, there is currently no practical approach to guarantee that user data is not misused. This paper fills this gap by enforcing user-defined data usage policies on the cloud and service providers. Unlike prior work that provides support for secure storage-free computations in custom systems [22,48], our aim is to preserve the compatibility with existing FaaS-based web applications. We show that the Function-as-a-Service (FaaS) paradigm with explicit (storage) interfaces and stateless functions provides an avenue for enabling stronger security guarantees for existing applications. We use this insight to build Glasswall, the first system that can protect user data without trusting either the cloud or function providers. We deploy Glasswall in OpenWhisk [8] to demonstrate a transparent integration into an existing FaaS platform.

Trust relationships in the cloud. It is common practice for a service provider to run its applications on the infrastructure of a cloud provider to serve requests from users. Ideally, these three parties (i.e., cloud provider, service provider, and user) do not have to trust each other; in particular, the user does not have to trust either party with their data, and the service provider does not have to trust a user not to hijack an instance or the cloud for not gaining access to its proprietary application. Existing systems have either relaxed the trust relationship by assuming that the service provider is trusted [21,33,53,15] or requiring a trusted fourth party to establish the security of the service provider's application [2,17,48]. Unfortunately, both these approaches have shortcomings: service providers are known to mishandle user data [50,36,51,23], and it is not practical to require a trusted party to perform repeated code audits for each version of the service provider's applications, or publish proprietary code for community auditing. The question we ask in this paper is whether it is possible to build a practical system in which the users do not have to trust either the cloud or service providers without requiring another party that vets the service provider, without limiting application expressiveness. We answer this question in the design of Glasswall, the first practical system with a strong user-defined data policy enforcement.

Information flow control. The user is the owner of the data and should hence define the data usage policies [26,21,46]. Glasswall enforces such policies using a strict Information Flow Control (IFC) mechanism that operates on the inputs and outputs of a given function. To ensure general functionality and preserve compatibility against the risk of over-tainting, Glasswall features two mechanisms for selective data declassification: (1) *permanent* declassification enables a user to reduce data usage restrictions (e.g., when sharing files), and (2) *transient* declassification for cases where a given function needs to (securely) operate on secret data (e.g., a password) without tainting its output.

Handling implicit state. While the IFC mechanisms in Glasswall enforce user-defined policies, a malicious function could still leak information *across* requests through implicit state inside the process that runs the function in naive implementations. Glasswall implements a lightweight and secure process *snapshot-and-resume* mechanism. On each request, Glasswall restores the function in a fresh process, eradicating any implicit process state with a small impact on performance of 5-10% over functions reusing a process for multiple requests.

Performance. A reference implementation of Glasswall introduces 72%/60%/62% overhead for I/O-intensive benchmarks on median latency/99th percentile latency/throughput compared to an unprotected baseline, respectively. Moreover, the reference implementation incurs only 4%-19% overhead for compute-only workloads, outperforming the state of the art. We show that Glasswall is the first practical system with end-to-end data protection for FaaS users.

Contributions. The following lists our contributions:

– We present the design of Glasswall, the first system that enforces user-defined data usage policies under a strong threat model with exchangeable components. The design of Glasswall features the following novel aspects:

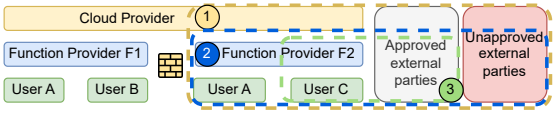


Fig. 1. Data usage boundaries in a FaaS environment. The gap between user-intended data usage and accessibilities of the cloud provider and function provider is not enforced in current state-of-the-art systems.

1. **Non-interactive secure protocol.** We design a protocol that enables secure VM instantiation and key distribution with an unknown function without interactivity requirements.
2. **Policy runtime.** We design and implement a runtime that enforces user-defined data usage policies and offers both permanent and transient declassification mechanisms.
3. **Lightweight process checkpoint/restore.** We implement a lightweight process-level checkpoint and restore mechanism implemented inside ELF directly, enabling efficient per-request restore by leveraging the limited FaaS process interface.

– We show that Glasswall's reference implementation [1] for OpenWhisk [8] using SEV-SNP [43] provides a practical solution for end-to-end data protection by evaluating using FaaS benchmarks.

2 Background

We provide some background on Function-as-a-Service (FaaS) and Trusted Execution Environments (TEEs), before motivating our work in Section 3.

2.1 Functions as a Service

Services offered on the internet reach millions of potential customers. In the traditional computing model, service providers would use multiple physical or virtual machines to provide load-balancing and maintain a high quality of service under high load. This slow and coarse scaling leads to undesirably slow responses or costs for unused resources.

FaaS or serverless computing differs from traditional cloud computing models by abstracting the infrastructure and runtime environment from the service developer. A cloud provider scales the number of function instances based on the incoming request load, enabling usage-based pricing rather than resource-based pricing [29]. Requests are finite messages to enable offline computation and resource requirement estimation. The platform stores requests in a database or message queue which feed into different function instances, buffering requests within the platform itself. A key differentiating factor compared to other modern, scalable approaches, such as microservices, is the absence of state in the

[1] https://github.com/comsec-group/glasswall.

function model. To enable running stateful applications with stateless functions, cloud providers offer tight integration with persistent data storage services [44]. FaaS supports diverse workloads, from batch to real-time processing [14].

To highlight the differences of a FaaS system, we showcase the transformation of a simple web application to the FaaS paradigm, and describe the flow of information through the system. Our example application "RemindMe" provides a service where users can register an alarm to be sent via email at a specified date. The application would be spread over a web server to host the website, a database to record the events, a polling service to poll for due events, and a mail server to send the reminders. Evidently, different workloads require scaling of different systems: more website visits, events entered, and reminders sent in a short timeframe put more stress on the web server, database, and mail server, respectively. In FaaS, the scaling of the HTTP frontend, database, and mail servers is up to the cloud provider and billed per-request (rather than by time of availability of resources). A `POST` request to register a reminder would be sent to the cloud provider, which puts the request into a coordinator queue for processing. The cloud provider starts or reuses an instance of the function handling the website, and passes the stored request to the instance. The function calls the cloud provider framework to register the event in the platform's database and returns the HTML for the success message to the cloud system. The FaaS platform puts it into the database for responses, from where the web frontend (which kept the connection open) picks up the response and sends it to the user. Instead of polling the database for events, the function also registers an alarm timer with the cloud provider, requesting the recorded database entry to be passed to another function at the specified time. Once this triggers, the cloud platform calls the function composing the email by passing the associated data to the coordinator queue. The function then retrieves the required information and sends the created email content to the platform, which then transmits the email. Crucially, if any of the components experience higher load, additional instances of the functions are created, and once load reduces, the platform shuts these instances down. In summary, the FaaS platform provides basic utilities as well as the scheduler and routing for event-driven functionality.

We depict the data usage boundaries in a FaaS environment in Figure 1. Both the cloud provider (①) and the function provider (②) have access to and can control data in less restrictive manners than the user intends (③). At the same time, the cloud provider does not enforce the function provider to only access data in the intended way. This also includes information flow between clients of the same function, which makes it very difficult to guarantee privacy for a single user. Namely, the model allows attackers to infer information about a previous request if they can access stale state. Even worse, both the function and the hypervisor could directly send user data to external parties which a user might disapprove.

2.2 Trusted Execution Environments

Trusted Execution Environments (TEEs) provide applications with varying levels of confidentiality and integrity depending on the implementation. Central to TEEs is the verification of code identity and its protection against inspection or manipulation by the host [42]. In Glasswall, we rely on SEV-SNP for the initial state attestation (verifying the initial state of the machine), and the ability to sign additional messages that enable verification of messages originating from machines with a given initial state. We explain the working principle of SEV-SNP in more detail in Appendix A. AMD SEV-SNP provides isolation to VMs by encrypting memory and locking VM state from direct access by the hypervisor [43].

3 Motivation

FaaS systems enable a new level of abstraction for developers: by chunking complex code bases into small, stateless functions and leaving scaling to the cloud provider, developers can focus on their business logic [4]. Although FaaS already offers significant benefits, we believe that there are opportunities to improve data security as well. We motivate the need for user-defined data flow policies based on three examples.

3.1 Motivational examples

(1) Hackers leaked sensitive data of roughly 150 million Americans at Equifax using a database data breach in 2017 [36]. (2) LastPass suffered a data breach in 2022 through a compromised employee account [27], leaking password archives which could be decrypted by brute-forcing the master passwords [25]. (3) Twitter notified users in 2018 about a security bug, exposing user clear-text passwords to Twitter employees through log data [50]. (4) In 2018, a hacker stole sensitive patient data of about 30,000 individuals from Vastaamo, a Finnish psychotherapy service provider [16], due to an overly open database. These security incidents all have in common that user data was accessible to a wider range of entities than what the data owner (implicitly) intended.

Despite efforts in data privacy laws, a service provider can still process data according to *legitimate interests*, which are not well-defined and subject to interpretation [34]. Even with explicit data usage prompts, unclear phrasing also leads to data use potentially not intended by a user, e.g. when using Apple's App Tracking Transparency (ATT) [32,54,35]. These examples and limitations of currently available technical and legal solutions to mitigate data leakage motivate a system where users can explicitly control how their data is used in the cloud without relying on unenforceable promises or fuzzy definitions.

Table 1. In our threat model, the TEE is a common trusted party, while there is mutual distrust between the user, function provider, and cloud provider.

	Trustee			
Trustor	User	Function Provider	Cloud Provider	TEE Vendor
User	id	○	○	●
Function Provider	○	id	○	●
Cloud Provider	○	○	id	●
TEE Vendor	○	○	○	id

4 Threat Model

We place the root of trust in the TEE as shown in Table 1. We consider the following parties: user, function provider, cloud provider, and TEE vendor. Consistent with threat models of similar work, we exclude timing and termination [22,17,48,15], network sampling [17,24,52,15] and microarchitectural side channels [22,2,48,52,15]. The first two cannot apply to arbitrary code without significantly impacting performance or functionality, the latter requires fixing in hardware for secure TEE implementations. With those exceptions in mind, the user trusts only the TEE vendor, but not the cloud provider, function provider, or other users. The function provider trusts the TEE vendor, and does not trust the user nor the cloud provider and considers their code a secret. The cloud provider does not trust the function providers and employs suitable isolation techniques against VM escapes to other tenants or the hypervisor itself. The threat model allows a function provider to run any code within the TEE, including interpreting data as code, without requiring (or enabling) any party to attest security or correctness of the application code. Similarly, we allow the cloud provider to attempt to completely observe, modify, and leak network messages passing through the system, in accordance with FaaS scaling mechanics [31]. Hence, we cover a wide range of attacks by function providers, cloud providers, and other users.

5 Glasswall Overview

We build Glasswall to bring back data control to the user. In this section, we discuss the requirements of building such a system (Section 5.1), and provide an overview of the high-level design and its challenges (Section 5.2).

5.1 Requirements

Appropriate use determination. The legitimate use of data is context-sensitive as exemplified in Section 3; only the data owner can determine the legitimate use of data reliably and consent to its usage, i.e., unlocking the data for a specific operation and actor.

R1: The user must control data usage explicitly.

This implies that a user needs to prepare and send additional information. On the flip side, it means that neither the cloud provider nor the function provider are burdened with the error-prone task of guessing the user's intent.

Enforcement. When a user provides data, they must be able to trust that the data usage terms are honored under all circumstances. In a potentially malicious environment, any malicious actor can simply remove the usage policy or replace it with a more permissive one.

R2: Both function provider and cloud provider must be controlled to honor explicit data usage policies.

As a consequence, data must be both encrypted and integrity protected, and no party must ever gain unchecked access to the data.

Implicit data flow. While information about legitimate data usage is a required part of the system, it is not sufficient. Aside from hijacked code, various performance optimizations, in particular software caching, can influence a later execution. A valid system must not allow such implicit flow of information.

R3: An invocation must not depend on state from a previous invocation.

Consequently, we must not rely on system-provided data-agnostic caches or other performance optimizations that may bypass the data usage policy.

Compatibility. Both retrofitting existing systems and designing new systems put a significant burden on the function developers, increasing the cost of development and hindering deployment [7]. To ensure that the system is deployable in existing systems and new code remains backward-compatible, we must not require changes to existing code and development practices by design. This does not mean that no code requires changes. For the compatibility of security, we require that no existing or new code can bypass the security guarantees within the threat model. In terms of *functionality*, the solution must not impede execution when sufficiently lax data usage policies are set.

R4: The system must be able to protect against data leakage of existing code bases.

Concretely, while user restrictions may break existing code which actually leaks data or where absence of data leaks cannot be guaranteed (false positive), code must not require changes if all its actions are permissible according to the user. Next, we will look at the high-level design and challenges of Glasswall.

5.2 High-Level Design and Challenges

We depict the high-level overview of Glasswall in Figure 2(a). Figure 2(b) illustrates the conceptual overview of a FaaS system augmented with Glasswall. We achieve **R1** (explicit data usage control) by enabling the user to define appropriate data usage policies within the request. The specified policies are enforced

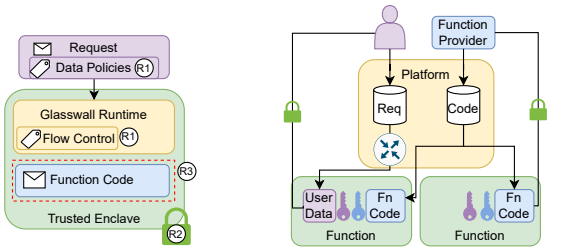


Fig. 2. Overview of a single function instance in Glasswall. The user supplies the appropriate use policies by tagging the request, which the Glasswall runtime handles in a data flow management engine (a). The entire system runs within a TEE to protect against a malicious cloud provider and to attest the loaded environment (b).

by the Glasswall runtime which filters the data flow passing into and out of the function. To ensure that the Glasswall runtime cannot be bypassed (**R2**, enforcement in adversarial environments), we rely on AMD SEV-SNP [43] in our reference implementation to provide a tamper-resistant environment. We ensure per-request isolation (**R3**, statelessness) by forcing a process isolation boundary between requests, preventing data flow through process-internal channels. We achieve the final requirement, **R4** (compatibility), through the design itself: All required modifications fit into existing FaaS designs.

The system design outlined in the previous paragraph shows a single instance preventing data from leaking once it has entered a function. As both the cloud provider and the function provider are considered untrusted in our threat model, Glasswall must additionally protect the data in transit in an end-to-end manner. However, the design of FaaS requires a cloud provider to distribute requests to arbitrary instances at an arbitrary time of execution, removing the interactive key exchanges with the sender from the set of possible solutions. We therefore require a way to protect the sensitive information exchanges between the function provider and the function instances (i.e., the function code) and between the users and function instances (request and response). The requirements of this system lead to three main challenges that Glasswall addresses with its key components: (1) *secure communication*, (2) *information leakage control* through the Glasswall runtime, and (3) efficient handling of the *implicit data flow*.

Secure communication. Our threat model assumes distrust between all parties, requiring authenticated encryption to provide confidentiality and integrity.

Ch.1 How can we ensure end-to-end secured communication with the intended parties? ⟶ *Section 6*

We solve this challenge by employing a trusted public key distribution system within the cloud provider to handle secure key distribution and identity attestation as detailed in Section 6.1.

Information leakage control. Glasswall allows the arbitrary untrusted functions to serve user requests, which introduces our second challenge:

Ch.2 How can we restrict the function code from leaking data without reducing its expressiveness? ⟶ *Section 7*

We address this in Section 7 by introducing primitives that enforce information flow without requiring application code analysis, along with declassification mechanisms for common tasks such as data sharing and user authentication.

Implicit data flow. By tackling the programmatic data flow, we cannot achieve complete security due to potential implicit data flows between requests that share the same function. A trivial approach to ensure a clean state between requests is to restart the process in which the function executes. This approach, however, is prohibitively expensive in terms of performance [30].

Ch.3 How can we reliably and efficiently prevent implicit data flow between requests? ⟶ *Section 8*

We solve this challenge by using native process snapshots and fast per-request resumes to ensure that no state carries over between requests without significantly impacting performance, as we discuss in Section 8.

6 Secure Communication

Glasswall is oblivious to the function code by using flow control mechanisms on inputs and outputs to prevent data leakage. This security property holds if only the entities that provide this data flow enforcement can decrypt the messages. Additionally, decryption must be able to happen offline, as a key property of FaaS is the ability to queue and schedule requests without user interaction. We achieve these properties by using a TEE to bundle long-term encryption keys in the published attestation report. The TEE vendor acts as the external root of trust for identifying secure contexts and providing an honest identifier of the initial memory and register state. Both user and function provider rely on the public attestation reports to determine secure systems. This is achieved by a Key Distribution Server (KDS), discussed next.

In the entire protocol, the cloud provider acts as a relay for messages and hosts the required resources in encrypted and authenticated virtual machines. It cannot decrypt encrypted data and any modification of signed data will be detected by the participating parties.

6.1 Key Distribution Server

TEE-based distributed systems like S-FaaS [2] solve the problem of secure keys by creating them within an SGX enclave and sharing the private keys only with the attesting parties. In this form, the attestation scheme requires system users

to have verified the code of components partaking in the handling of a particular request. Function providers, however, may want to keep their code secret from the cloud provider. We show how Glasswall removes the requirement for the user to attest individual function instances and further enables the secrecy of code with its KDS. The KDS is part of the attested components (Trusted Computing Base) of Glasswall and runs inside an attested enclave.

The KDS has three tasks: key creation, key distribution, and attestation verification. We discuss the protocol steps next.

6.2 Protocol Steps

We show the message sequence chart in Appendix B, Figure 7. We next describe the steps of the protocol.

(1) Function registration. When a function is registered with the cloud provider, the cloud provider requests the KDS to create, attest, and publish two function-specific key pairs, namely the function code key and the function transport key. The function provider then demands the cloud provider to send the set of attested keys before encrypting their function image with the code key and uploading it. This ensures that the cloud provider cannot inspect the function image which can contain keys or trade secrets [18]. Lastly, the function provider signs the public part of the function transport key and publishes it for the user, confirming that the function provider has verified the platform and is the author of this function.

(2) User communication. A user sends their sensitive data after verifying that the transport keys have been created by the KDS and belong to the intended function (2a). To do this, the user first verifies the attestation report published by the KDS when it receives the public function transport key. Additionally, the user checks if the specified function belongs to the intended function provider by verifying the signature of the function transport key. Subsequently, they encrypt the payload with the function-specific transport key before sending the request to the cloud provider for handling (2b). Note that the client must only obtain the chip-specific TEE keys of the KDS once, and can reuse them for all functions managed by the KDS. Other Glasswall-enabled request senders (e.g. databases with timed triggers) follow the same scheme.

(3) Function execution. When the FaaS platform receives a request to run a function for the first time, it starts a new function instance within a TEE (3a). The VM creates its own key pair, binds it to its attestation report, and sends it to the KDS. After sending its initial state and a hash of the runtime image, it will continue to load the language-specific runtime (e.g., Python), initialize the handlers, and start accepting requests from the platform. Once the KDS has attested the authenticity of the instance, it responds with the code and transport keys. When the instance receives these keys, it decrypts and installs the function code, and starts decrypting and processing the incoming requests (3b).

Cold-start cost. The steps (1), (2a), and (3a) need to be executed on initialization of the respective participants (cold start). The Glasswall protocol adds

Table 2. Asserted protocol properties. Request agreement requires user authentication, and injective function code agreement is not required for security.

Element	Secrecy	Agreement (non-inj.)	Agreement (inj.)
Request (user → *function*)	●	○	○
Response (function → user)	●	●	●
Function code (provider → instance)	●	●	N/A

additional cost on communication and signature verification in these steps. For subsequent (warm) requests, Glasswall only requires additional overheads for encryption and authentication. We minimize attestation costs by removing the interactivity requirements, enabling secure caching and reuse of keys.

6.3 Security

The secrets never leave the defined environment of authorized KDS machines because KDS VMs are not accessible by either the hypervisor or function providers, and they do not contain code that leaks the secret keys to non-attested VMs.

Properties. Trivially, we want to achieve confidentiality on the secrets passing through the system. Additionally, a client needs to be sure that messages actually reach the intended target (and no MitM), and that all responses originate from the correct function.

Using the definition of Lowe [28], we prove the absence of MitM attacks (non-injective agreement) and replay attacks on the response (injective agreement).

Tamarin verification. We have defined a Tamarin [9] model of the KDS assuming a secure TEE and obtained proofs for all required confidentiality and integrity properties. In summary, a cloud provider cannot pass a malicious KDS because both function provider and user will not accept the attestation report. Similarly, the cloud provider cannot pass a malicious runtime to the function container, as the KDS will not send the secret transport keys to unattested entities. Sending tampered function code causes an error during authenticated decryption. Lastly, if a malicious function provider tries to leak all data, the Glasswall runtime will prevent leaks which violate the data usage policies. We will discuss the details of the runtime and data usage policies in the next section.

7 Glasswall Runtime

As discussed, Glasswall does not inspect or restrict the function code in the attestation process. Instead, the Glasswall runtime provides an environment that enforces data usage policies which the user associates with their requests. System-level facilities, such as DNS, are handled by the OS within the secured context. The Glasswall runtime enforces policies like DNSSEC.

7.1 User-Defined Policies

Glasswall uses data policy tags to associate user intent with the data itself. In our threat model, we do not know which function is operating on the data, or

Table 3. Data policy tags implemented in our prototype. Glasswall allows for arbitrary policy tags, where the semantics are defined by the Glasswall runtime.

Policy Tag	Description
`NoOutput`	This data must never leave the function.
`NoPersistence`	This data must not be stored persistently.
`NoExternalService`	All flows must be contained within the system.
`NoMixing`	No foreign data must enter the function.
`NoListing`	Treat the object as non-existent.
`DebugReport`	Do not enforce policies, but report violations.
`Unlock`	Disregard policies with matching UIDs.

for what purpose. Consequently, we do not rely on currently ubiquitous actor (function) permissions [5], but rather add restrictions to data objects, and let the actors inherit the restrictions from the data they touch. We now discuss the policy system in detail.

Policies. Users determine policies based on the context of data use, supported by automatic heuristics. As an example, the user can attach a policy that ensures that their password is not permanently stored by the system. A policy is an abstract notion of how a user intends data to be used. Namely, the policy specifies which predicate function should determine if a given potential flow out of the function is legal. The available predicate functions depend on the implementation of the Glasswall runtime. Glasswall operates based on the user providing a set of policies in a machine-readable form (*policy tags*) along with the data sent to a service, indicating the intended (and therefore legitimate) use of the data.

Table 3 shows a set of policy tags and the associated policy semantics that are currently implemented by the Glasswall runtime. To give some examples, `NoExternalService(+example.com)` blocks connections to external domains except for `example.com`, and `NoOutput` requires that the function does not produce any output once it consumes data with such a tag (e.g., a password). We emphasize that domain-specific policies can be enabled in any Glasswall implementation, enabling arbitrarily complex rules at the cost of increased verification complexity of the Glasswall runtime. The combination of all input policies propagate to all outputs, *tainting* them in the process. In addition to the list of allowed exceptions to the rules, each policy tag can additionally take an unlock identifier (i.e., a `UID`) that is only known to the user. Glasswall allows each data object to be associated with multiple policy tags with different `UID`s. The modifier `Unlock(UID)` tag allows the user to remove policy tags that are associated with a given `UID`. We show how the `Unlock` tag provides the users with a declassification mechanism in Section 7.2.

Policy Enforcement. The Glasswall runtime enforces the policies specified by the user by intercepting operations that may have effects outside the current process context. Since system calls provide the boundary between the process context and system-internal or external resources, we prevent system calls from the function through `seccomp`. Instead, we use a `socketpair`-based IPC channel to pass requests to the Glasswall runtime which then executes the system calls if the policies allow it before returning the result. To reduce overhead, the Glass-

wall runtime offers the functions simplified interfaces, called *informed* interfaces, that offload complex operations such as object storage access or HTTPS connections. *Informed* interfaces allow applications to send a single message to the runtime, describing a high-level action (e.g. an HTTP POST operation), which then handles connection pooling, encryption, and buffers, providing a stateless interface for otherwise stateful operations. Since it is common for FaaS platforms to provide optimized interface libraries, the function code does not have to be adjusted for informed interfaces, enabling transparent integration.

Policy Engine. The policy engine is the part of the Glasswall runtime that masks external effects by tracking external resources and transparently removing and adding policies on input and output, respectively. To achieve this, it keeps track of two sets during the invocation lifetime of a function:

1. *Current policies set*: this is a set of active policies which keeps the union of the policies of data objects seen by the function at any given point.
2. *Current requirements set*: this is a set of active leakage channels at a given point created by external effects (e.g., network connections). The Glasswall runtime derives *requirements* from system calls and informed interfaces and tracks currently active *requirements* to monitor long-lived connections.

To enforce current policies of data objects, Glasswall denies an operation if a requirement of the operation conflicts with current active policies (i.e., the current policies set). For example, when a function tries to open a TCP connection, it *requires* the runtime to open an information channel to the given address. The runtime then derives the generalized *requirement*; for TCP connections, this *requirement* contains the target domain. If the new *requirement* conflicts with currently active policies, the operation is denied.

Input/Output handling. To prevent violating policies of new data objects, Glasswall refuses to forward incoming data objects when their policies conflict with current requirements of open connections (i.e., the current requirements set). Furthermore, when a function accesses data objects over the informed interfaces, the policy engine obtains and strips policy information from the incoming data before forwarding it to the function, and updates the current policies set accordingly. Similarly, when data leaves the function over informed interfaces, the system attaches the current policies set to the outgoing message.

Walk-through example. We show an example invocation in Figure 3. Whenever an interface is used, its implications are loaded into the requirement set and checked against the current policies. In the example, the first connection does not violate the initial policies, so it passes. However, the system stops a subsequent database request where the attached policy denies external service access. After the function closes the connection, a data load can pass through. However, once the data is accepted, the attached data policy sticks and future connections to the external service will be denied. In summary, the system generates requirements from the execution and enforces the policies on these requirements during the lifetime of the function invocation.

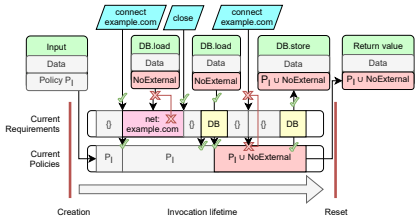


Fig. 3. The policy engine tracks the requirements of external effects and checks them against the current policies to guarantee that the policies are honored. Both raw system calls (blue) and informed operations (green) are supported.

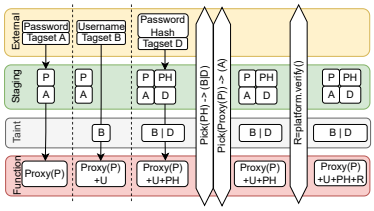


Fig. 4. Transient declassification: the password is tagged as `NoOutput`; the runtime avoids secret-dependent loads by not forwarding it to the function and performs the comparison internally.

7.2 Permanent Declassification

Traditionally, trusted declassifier functions are used to remove taint from information flows that are provably safe [38], for example, when a secret key is used in a cryptographic operation. However, because we only observe external effects and remain oblivious to the function code, we cannot know if a function uses a secret to encrypt data or if leaks the secret. At the same time, a function must not be able to declassify and leak data by itself without the user explicitly allowing the operation.

When a user wants to permanently declassify a data object, they can use the `Unlock` tag by providing the `UID` of the policy that they want to remove from the data object. The Glasswall runtime processes this special *modifier* policy tag by finding and removing the policy that matches with the `UID`. While this enables user-controlled declassification, it is not suitable for cases where the legitimacy of a declassification depends on the function code. As an example, if a user wants to allow a rating agency to share their credit data with a bank, they can associate their credit data with non-sharing requirements, except to the bank in question (i.e., `NoExternalService(+bank.com)`). However, this data is now permanently accessible by the bank, since they could copy the data to a different system while analyzing it. Therefore, we consider this approach to be permanent declassification, even though the data can be reclassified at a later point by providing a more restrictive policy tag.

7.3 Transient Declassification

Permanent declassification is not desirable when long-term secrets are mixed into a calculation. Such secrets include cryptographic materials, access tokens or passwords, which cannot be declassified permanently, but need to be loaded to perform a verification or encryption. Under strict policy checking, this would break existing code. Concretely, if a function performs a permission check before executing, all outputs would inherit the highly restrictive policies of a secret.

We therefore offer a limited platform-based declassification approach that allows some operations to execute securely as discussed next.

Platform staging. Figure 4 shows how we provide a platform-based transient declassification. The Glasswall runtime includes a set of pre-defined functions for secure password checks, hash functions, and cryptography. Functions that aim to do transient declassification indicate to the runtime that it should not pass the loaded value to the function directly (which would also taint the function). As a result, selected inputs to the function are converted into proxy objects that reference a value that is staged within the runtime but content-less in the function. Because neither errors nor values are passed to the function code, proxy objects do not change the taint of an execution. The function code can then specify a comparison operation on the proxy values and request the result from the runtime. Since the secret values are not visible to the function code, this comparison must be blind. That is, both the selection values to compare and the comparison function cannot depend on a secret, which is enforced by the taint tracking. **Separation of data.** Glasswall becomes restrictive when a function consumes many inputs with different policies. This enforces best-practices for secure applications which already stipulates a clear separation of secret and non-secret data to prevent accidental leakage through coding errors.

8 Handling Implicit Data Flows

In a truly stateless environment, secure user-function communication through the Glasswall runtime is sufficient to achieve secure user data processing. However, even stateless code uses memory as a scratchpad, and may employ explicit caching techniques which may keep state between requests. Consequently, information may leak across requests through stale or cached data. A prominent example of when stale memory leaks information is described in [10], where an HTTP caching proxy would accidentally leak memory from other requests due to a bug in the parser state management. Glasswall leverages the simplified FaaS process interface to achieve a lightweight per-request process startup as we describe next.

8.1 Process-level Per-request Isolation

The overhead of starting a new process for every request can vary greatly depending on the size of the binary and the function-internal initialization. For FaaS, initialization is generally the most expensive part. To skip the initialization, it is possible to utilize fork-based snapshot/resume, where a base process acts as a template for new processes which then handle the user requests [1,13]. This approach is incomplete as shared mappings and file descriptors are not cleared and threads are not preserved [11].

To address these issues, Glasswall relies on a snapshot mechanism that is restored inside a new process for every request. To build a snapshot, Glasswall

first runs public test cases back-to-back to an unmodified function to ensure that the instance has loaded all of the required dependencies into its address space.

Handling kernel state. In FaaS systems, function binaries frequently use a pipe-based interface to communicate with a handler process. In our implementation, inputs are passed on the standard input pipe, and the standard output and error streams are logged. We use an additional file descriptor to return the results to the handler. Glasswall introduces another file descriptor to communicate with the Glasswall runtime, handling system calls that would otherwise be filtered. Because of this fixed interface, we do not have to manage file descriptors and kernel-side state in the function snapshot. Temporary files are handled by the runtime, which stores the files within the processes' address space. This ensures that all changed data is automatically cleared without further intervention.

Introduced reproducibility. The state rollback resulting from the snapshot restoration breaks pseudo-randomness, but not randomness freshly obtained from the kernel interface. While this breaks the security of internal cryptography, these internal implementations are not considered secure by Glasswall in the first place, as they can include other weaknesses.

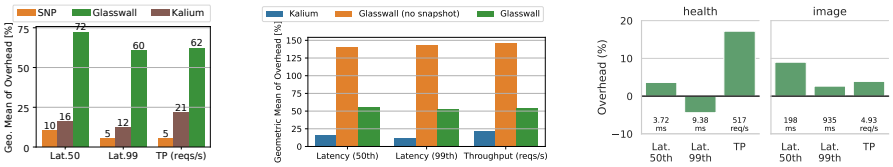
9 Evaluation

We compare the end-to-end performance of Glasswall with Kalium [24] which is the state of the art that provides function-level data flow integrity but trusts the function and cloud providers (Section 9.1). We port the Nordstrom *Hello Retail* applications used by other solutions [12,24] to Python to run OpenWhisk [8] which the Glasswall reference implementation builds upon. We report on the geometric mean of overhead over all the benchmarks.

Evaluation platform. We use a platform based on AMD EPYC 7443 for evaluating snapshot/resume and another based on AMD EPYC 7413 for measuring the end-to-end performance on a localhost connection. We use Ubuntu 22.04 with AMD's SNP-enabled kernel 6.9-rc2.

9.1 End-to-end performance

Figure 5a shows the latency of Glasswall and compares the relative overhead of median latency, 99th percentile latency, and throughput of the complete Glasswall platform with Kalium [24] relative to unprotected baselines. We calculate the overhead relative to an OpenWhisk instance with QEMU-based function runners for Glasswall and a gVisor-based [19] OpenFaaS setup for Kalium[24]. Compared to Kalium, Glasswall incurs a higher overhead while additionally protecting against malicious hypervisor and function code. Namely, Glasswall introduces an overhead of 72%/60%/62% on median latency/99th percentile latency/throughput and Kalium introduces an overhead of 16%/12%/21%, respectively. We explain the results based on three key factors:



(a) Hello Retail (b) Glasswall overhead (c) Ryoan benchmarks

Fig. 5. IO-heavy workloads like *Hello Retail* incur an overhead of 72% in the median latency (a). Compute-bound workloads as used in the SotA [22] (b) Snapshots provide significant performance gain (overhead relative to SNP-baseline) (c) show lower overhead compared to IO-heavy workloads. The reduction in 99th percentile latency is due to the snapshots providing a more deterministic interface with fewer outliers.

1. Function duration. The *HelloRetail* benchmark serves HTTP requests, often in less than 40 milliseconds end to end. For these short-running functions, process isolation and context switching become dominant factors.

2. Database operations. In the benchmark, we use Minio as a storage backend. The policy checks required by Glasswall need to be dispatched as separate requests in some cases, e.g., when an object is listed, but not actually fetched. This increases the overhead in database-heavy functions.

3. Snapshot overhead reduction. While Glasswall is always faster than Glasswall without snapshots due to skipping runtime initialization, some benchmarks show performance benefits over the baseline. This happens in code that inefficiently (but correctly) creates a new database connection on every request, therefore not being able to reuse pipelining. Glasswall re-enables connection reuse at the runtime level, improving performance of idiomatic code in a secure manner.

Cold start and scaling. We exclude the one-time key setup and instance creation. While SEV- and dm-verity protection shows lower cold start performance for the generic QEMU implementation, other work like *SEVeriFast*[20] shows that it is possible to reduce the overhead to less than 100ms.

9.2 Comparison with side-effect free code

Compared to Glasswall, Ryoan [22] offers less flexibility, blocking system calls altogether after a first initialization because it defines a static set of rules rather than tracking the permissions of data flowing through the system. While Glasswall supports a wider range of functionality, the benchmarks used by Ryoan make no database accesses or other side effects take place at runtime. We evaluated the overheads of Glasswall relative to an unprotected baseline on the same benchmarks used by Ryoan. The results in Figure 5c show that Glasswall provides significantly lower overheads for heavily compute-bound tasks compared to the numbers reported in [22] (4% overhead compared to 24%, 19% compared to 419% in the best and worst cases, respectively).

Table 4. Comparison of Glasswall with related work. Code-agnostic approaches do not require knowledge about the code.

	Project	User-def.	Code Secrecy	Data Secrecy	Persist.	Code agn.	SbD/ Flex	Platform	Robust B	A	C	S	Overhead
Online	Glasswall	●	Ⓤ Ⓒ	Ⓕ Ⓒ	●	●	●/●	OpenWhisk	●	●	●	●	4-62%
	Ryoan [22]	◐*	Ⓤ Ⓒ	Ⓕ Ⓒ	○	●*	●/○	Custom	●	●	●	●	27-419%
	Kalium [24]	○	-	-	○	○	○/●	OpenFaaS	●	●	○	○	21%
	WebTTC [21]	●	-	-	●	○	●/○	Custom	●	○	○	○	0-150%
	Trapeze [3]	○	-	-	●	○	○/●	AWS	●	●	○	○	52-92%
	Valve [12]	○	-	-	◐	○	○/●	OpenFaaS	●	●	○	○	150%
	LeakLess [39]	○	-	-	●	○	○/●	Spin	●	●	○	○	8.5%
Offline	S-FaaS [2]	○	-	Ⓕ Ⓒ	○	○	●/○	Custom	●†	●†	●	●	3.4%
	TAPDance [23]	○	-	Ⓕ Ⓒ	◐	○	●/○	Custom	●†	●†	●	●	-33%***
	Revelio [17]	○	-	Ⓕ Ⓒ	●	○	●/●	Custom	●†	●†	●	●	15%
	Clemmys [48]	○	-	Ⓕ Ⓒ	○	○	●/○	OpenWhisk	●†	●†	●	●	10%
	SplitContainer [45]	○	-	Ⓒ	●	○	●/●	Custom	◐	●	●	○	7.8%
	RiverBed [52]	●	-	-	●	○**	○/●	Custom	●	◐	○	○	10%
	PrivGuard [53]	●	-	-	●	○	○/●	Custom	●	○	○	○	10s
	RuleKeeper [15]	○	-	-	●	○	○/●	Custom	●	◐	○	○	10-20%
	WILLIAM [41]	○	-	-	●	○	○/●	OpenFaaS	●	●	○	○	0.5%
	CloudFlow [37]	○	-	-	●	○	○/●	AWS	●	●	○	○	0%

* User must know the execution DAG, recompilation required
** Relies on an additional trusted party for attestation
*** Side effect from precompiling typescript code
† Users audit the code
Ⓤ Ⓕ Ⓒ Not accessible/obtainable by the user, function, and cloud provider
Ⓤ Ⓕ Ⓒ Not accessible in detail, but leaks structural information

●/● Secure by design, allows lower security on consent
●/○ Secure by design, disallows insecure configurations
○/● Allows insecure configurations without user consent
B Bugs (accidental misconfiguration)
A Attackers (arbitrary code execution)
C Cloud Provider attacks
S Malicious Service/Function provider

10 Related work

Table 4 shows a comparison of Glasswall with related work.

User-defined access control. User-defined access control allows a user to specify how their data may be used. Most academic proposals rely on manifest-based access control, relying on the service provider to enforce the policies [15,24,41,12,3]. Some systems rely on different policy languages [21,53], while others rely on different policies for different users [52]. Split containers[45] does not implement policies. Glasswall attaches user-defined policies to data instead of actors.

Code-agnostic approaches. Certain systems rely on policies derived from the running code [15,53,24,41,12] or require code to be written in a domain-specific language [21]. The generic TEE-based approaches [17,2,23,45] derive security from the attestation of the software itself. Glasswall follows the code-oblivious, runtime-based approach without relying on the security of the function code.

Persistence. Securely storing data is challenging because of the key management involved. Most TEE-based approaches cannot support persistence because they require the client or an enclave to interactively provide encryption and decryption capabilities on ephemeral keys [22,48,2]. Full-system isolation approaches [17,45] provide storage within the trust domain and are secure as well. Most other solutions do not consider data security of persistent data.

Security by design and flexibility. Many security solutions suffer from limited adoption due to pre-existing code and workloads that are incompatible with

the new security requirements (secure by design, but not flexible) [22,2,23,48]. On the other hand, flexible solutions rely on potential exceptions or explicit deny-lists to provide functionality, but are not secure by design [24,3,12,39,52,53,15,41,37]. The user-defined properties of Glasswall allow security by design (blocking everything that violates the policies) while keeping the flexibility of running insecure functions when a user does not require secure execution. Other systems achieving the same properties use a reduced threat model [17,45], e.g., they do not keep the application code confidential by requiring code attestation.

Robustness. We differentiate four levels of data disclosure risks: bugs (accidental leakage), attacks (code injection), cloud provider inspection, and service provider leakage. *Bugs* refer to coding errors, where an honest developer accidentally implements incorrect permission checking, which is solved by auditing code or blocking unintended effects [17,2,48,52]. In contrast, all other solutions allow for bugs to occur, but block their effects if they violate a usage policy. For *Attackers*, we assume that there is a vulnerability that leads to code execution in the function. Solutions using in-process isolation [21] are vulnerable to such attacks [11,40], in contrast to solutions using process isolation [49,45]. *Malicious cloud providers* and *service providers* are surprisingly rarely included in threat models of prior work [45]. Glasswall protects user data in all these scenarios.

Platform. We list the platform for completeness. *Custom* systems include both FaaS and traditional server-based systems. Custom systems based on TEEs frequently divert from the store-and-forward approach of FaaS systems where the request passes a coordinator, reducing the coverage of FaaS use cases.

Overhead. We list the self-reported overhead for execution, excluding one-time setup costs. High overhead is generally caused by privacy checks occurring on the critical execution paths (*Online*). Glasswall provides overhead comparable to other dynamic information flow checking techniques under a stronger threat model. *Offline* verification includes solutions where the general security is proven offline (e.g. code attestation by the user or by the function provider itself). *Offline* solutions require no or very limited filtering at runtime.

11 Conclusion

We presented Glasswall, a new FaaS system compatible with existing platforms that enforces user-defined data usage policies without trusting either the cloud or function providers. Glasswall achieves this with TEEs and a key distribution scheme. To protect user data against malicious function providers and cleanse implicit process states, Glasswall features a policy-enforcing runtime and utilizes lightweight, efficiently resumed process snapshots per request. Our evaluation shows that the reference implementation of Glasswall introduces 72% and 60% overhead on latency and throughput, respectively, while providing protection without trusting either the cloud or function provider.

Acknowledgments. This work was supported by the Swiss State Secretariat for Education, Research and Innovation under contract number MB22.00057 (ERC-StG PROMISE).

References

1. Akkus, I.E., Chen, R., Rimac, I., Stein, M., Satzke, K., Beck, A., Aditya, P., Hilt, V.: SAND: Towards High-Performance serverless computing. In: 2018 USENIX Annual Technical Conference (USENIX ATC 18). pp. 923–935. USENIX Association, Boston, MA (Jul 2018), https://www.usenix.org/conference/atc18/presentation/akkus
2. Alder, F., Asokan, N., Kurnikov, A., Paverd, A., Steiner, M.: S-faas: Trustworthy and accountable function-as-a-service using intel sgx. In: Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop. p. 185–199. CCSW'19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3338466.3358916, https://doi.org/10.1145/3338466.3358916
3. Alpernas, K., Flanagan, C., Fouladi, S., Ryzhyk, L., Sagiv, M., Schmitz, T., Winstein, K.: Secure serverless computing using dynamic information flow control. Proc. ACM Program. Lang. **2**(OOPSLA) (oct 2018). https://doi.org/10.1145/3276488, https://doi.org/10.1145/3276488
4. Amazon: Aws lambda (2024), https://web.archive.org/web/20240525202604/https://aws.amazon.com/lambda/, accessed: 2024-05-25
5. Amazon: Managing permissions in aws lambda (2024), https://web.archive.org/web/20241113082048/https://docs.aws.amazon.com/lambda/latest/dg/lambda-permissions.html, accessed: 2024-11-13
6. AMD: Securing linux vm boot with amd sev measurement (2021), https://web.archive.org/web/20240226044558/https://static.sched.com/hosted_files/kvmforum2021/ed/securing-linux-vm-boot-with-amd-sev-measurement.pdf, accessed: 2024-02-26
7. Amron, M.T., Ibrahim, R., Bakar, N.A.A., Chuprat, S.: Determining factors influencing the acceptance of cloud computing implementation. Procedia Computer Science **161**, 1055–1063 (2019)
8. Apache: Openwhisk (2024), https://openwhisk.apache.org/
9. Basin, D., Cremers, C., Dreier, J., Sasse, R.: Symbolically analyzing security protocols using tamarin. ACM SIGLOG News **4**(4), 19–30 (2017)
10. Cloudflare: Incident report on memory leak caused by cloudflare parser bug (2024), https://web.archive.org/web/20240520052340/https://blog.cloudflare.com/incident-report-on-memory-leak-caused-by-cloudflare-parser-bug/, accessed: 2024-05-20
11. Connor, R.J., McDaniel, T., Smith, J.M., Schuchard, M.: {PKU} pitfalls: Attacks on {PKU-based} memory isolation systems. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 1409–1426 (2020)
12. Datta, P., Kumar, P., Morris, T., Grace, M., Rahmati, A., Bates, A.: Valve: Securing function workflows on serverless computing platforms. In: Proceedings of The Web Conference 2020. p. 939–950. WWW '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3366423.3380173, https://doi.org/10.1145/3366423.3380173
13. Du, D., Yu, T., Xia, Y., Zang, B., Yan, G., Qin, C., Wu, Q., Chen, H.: Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. p. 467–481. ASPLOS '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3373376.3378512, https://doi.org/10.1145/3373376.3378512

14. Eismann, S., Scheuner, J., van Eyk, E., Schwinger, M., Grohmann, J., Herbst, N., Abad, C.L., Iosup, A.: A review of serverless use cases and their characteristics (2021)
15. Ferreira, M., Brito, T., Santos, J.F., Santos, N.: Rulekeeper: Gdpr-aware personal data compliance for web frameworks. In: 2023 IEEE Symposium on Security and Privacy (SP). pp. 2817–2834 (2023). https://doi.org/10.1109/SP46215.2023.10179395
16. France-Presse, A.: "'Shocking' hack of psychotherapy records in Finland affects thousands". The Guardian (archived) (Oct 2020), https://web.archive.org/web/20250617184752/https://www.theguardian.com/world/2020/oct/26/tens-of-thousands-psychotherapy-records-hacked-in-finland, accessed 17 June 2025 (archived)
17. Galanou, A., Bindlish, K., Preibsch, L., Pignolet, Y.A., Fetzer, C., Kapitza, R.: Trustworthy confidential virtual machines for the masses. In: Proceedings of the 24th International Middleware Conference. p. 316–328. Middleware '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3590140.3629124, https://doi.org/10.1145/3590140.3629124
18. GitGuardian: The state of secrets sprawl report 2023 (2023), https://web.archive.org/web/20240118161250/https://www.gitguardian.com/files/the-state-of-secrets-sprawl-report-2023, accessed: 2023-01-18
19. Google: gVisor performance (2023), https://web.archive.org/web/20240105191409/https://gvisor.dev/docs/architecture_guide/performance/, accessed: 2023-01-05
20. Holmes, B., Waterman, J., Williams, D.: Severifast: Minimizing the root of trust for fast startup of sev microvms. In: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. pp. 1045–1060 (2024)
21. Hublet, F., Basin, D., Krstić, S.: User-controlled privacy: taint, track, and control. Proceedings on Privacy Enhancing Technologies (2024)
22. Hunt, T., Zhu, Z., Xu, Y., Peter, S., Witchel, E.: Ryoan: A distributed sandbox for untrusted computation on secret data. ACM Trans. Comput. Syst. **35**(4) (dec 2018). https://doi.org/10.1145/3231594, https://doi.org/10.1145/3231594
23. Jegan, D.S., Swift, M., Fernandes, E.: Architecting trigger-action platforms for security, performance and functionality. In: Network and Distributed System Security (NDSS) Symposium (2024)
24. Jegan, D.S., Wang, L., Bhagat, S., Swift, M.: Guarding serverless applications with kalium. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 4087–4104. USENIX Association, Anaheim, CA (Aug 2023), https://www.usenix.org/conference/usenixsecurity23/presentation/jegan
25. Krebs, B.: Experts fear crooks are cracking keys stolen in lastpass breach (2023), https://krebsonsecurity.com/2023/09/experts-fear-crooks-are-cracking-keys-stolen-in-lastpass-breach/, accessed: 2023-09-01
26. Kuhnhauser, W.E.: A paradigm for user-defined security policies. In: Proceedings. 14th Symposium on Reliable Distributed Systems. pp. 135–144. IEEE (1995)
27. LastPass: Notice of recent security incident (2022), https://web.archive.org/web/20240529072111/https://blog.lastpass.com/posts/2022/12/notice-of-recent-security-incident, accessed: 2024-05-29
28. Lowe, G.: A hierarchy of authentication specifications. In: Proceedings 10th computer security foundations workshop. pp. 31–43. IEEE (1997)

29. Lynn, T., Rosati, P., Lejeune, A., Emeakaroha, V.: A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms. In: 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). pp. 162–169. IEEE (2017)
30. Manner, J., Endreß, M., Heckel, T., Wirtz, G.: Cold start influencing factors in function as a service. In: 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion). pp. 181–188. IEEE (2018)
31. Manner, J., Wirtz, G.: Resource scaling strategies for open-source faas platforms compared to commercial cloud offerings. In: 2022 IEEE 15th International Conference on Cloud Computing (CLOUD). pp. 40–48. IEEE (2022)
32. Mohamed, R., Arunasalam, A., Farrukh, H., Tong, J., Bianchi, A., Celik, Z.B.: ATTention please! an investigation of the app tracking transparency permission. In: 33rd USENIX Security Symposium (USENIX Security 24). pp. 5017–5034. USENIX Association, Philadelphia, PA (Aug 2024), https://www.usenix.org/conference/usenixsecurity24/presentation/mohamed
33. Muthukumaran, D., O'Keeffe, D., Priebe, C., Eyers, D., Shand, B., Pietzuch, P.: Flowwatcher: Defending against data disclosure vulnerabilities in web applications. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. p. 603–615. CCS '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2810103.2813639, https://doi.org/10.1145/2810103.2813639
34. Niedermeier, R., Mpame, M.E.: Processing personal data under article 6 (f) of the gdpr: The concept of legitimate interest. Int'l J. Data Protection Officer, Privacy Officer & Privacy Couns. **3**, 18 (2019)
35. Ogut, A., Turanlioglu, B., Metiner, D.C., Levi, A., Yilmaz, C., Çetin, O., Uluagac, S.: Dissecting privacy perspectives of websites around the world: "aceptar todo, alle akzeptieren, accept all...". In: 33rd USENIX Security Symposium (USENIX Security 24). pp. 2849–2863. USENIX Association, Philadelphia, PA (Aug 2024), https://www.usenix.org/conference/usenixsecurity24/presentation/ogut
36. Primoff, W., Kess, S.: The equifax data breach: What cpas and firms need to know now. The CPA Journal **87**(12), 14–17 (2017)
37. Raffa, G., Blasco, J., O'Keeffe, D., Dash, S.K.: {CloudFlow}: Identifying security-sensitive data flows in serverless applications. In: 34th USENIX Security Symposium (USENIX Security 25). pp. 1073–1090 (2025)
38. Rocha, B.P., Conti, M., Etalle, S., Crispo, B.: Hybrid static-runtime information flow and declassification enforcement. IEEE Transactions on Information Forensics and Security **8**(8), 1294–1305 (2013)
39. Rostamipoor, M., Ghavamnia, S., Polychronakis, M.: Leakless: Selective data protection against memory leakage attacks for serverless platforms. In: Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA (2025)
40. Saito, T., Watanabe, R., Kondo, S., Sugawara, S., Yokoyama, M.: A survey of prevention/mitigation against memory corruption attacks. In: 2016 19th International Conference on Network-Based Information Systems (NBiS). pp. 500–505. IEEE (2016)
41. Sankaran, A., Datta, P., Bates, A.: Workflow integration alleviates identity and access management in serverless computing. In: Proceedings of the 36th Annual Computer Security Applications Conference. p. 496–509. ACSAC '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3427228.3427665, https://doi.org/10.1145/3427228.3427665

42. Schneider, M., Masti, R.J., Shinde, S., Capkun, S., Perez, R.: Sok: Hardware-supported trusted execution environments. arXiv preprint arXiv:2205.12742 (2022)
43. Sev-Snp, A.: Strengthening vm isolation with integrity protection and more. White Paper, January **53**, 1450–1465 (2020)
44. Shahrad, M., Balkind, J., Wentzlaff, D.: Architectural implications of function-as-a-service computing. In: Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture. pp. 1063–1075 (2019)
45. Shi, J., Gu, J., Xia, Y., Chen, H.: Serverless functions made confidential and efficient with split containers. In: 34th USENIX Security Symposium (USENIX Security 25). pp. 1091–1110 (2025)
46. Simpson, A.: On the need for user-defined fine-grained access control policies for social networking applications. In: Proceedings of the workshop on Security in Opportunistic and SOCial networks. pp. 1–8 (2008)
47. Tianocore: Ovmf (2024), https://github.com/tianocore/tianocore.github.io/wiki/OVMF
48. Trach, B., Oleksenko, O., Gregor, F., Bhatotia, P., Fetzer, C.: Clemmys: towards secure remote execution in faas. In: Proceedings of the 12th ACM International Conference on Systems and Storage. p. 44–54. SYSTOR '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3319647.3325835, https://doi.org/10.1145/3319647.3325835
49. Tsai, C.C., Arora, K.S., Bandi, N., Jain, B., Jannen, W., John, J., Kalodner, H.A., Kulkarni, V., Oliveira, D., Porter, D.E.: Cooperation and security isolation of library oses for multi-process applications. In: Proceedings of the Ninth European Conference on Computer Systems. pp. 1–14 (2014)
50. Twitter: Keeping your account secure (2018), https://web.archive.org/web/20240315041626/https://blog.twitter.com/official/en_us/topics/company/2018/keeping-your-account-secure.html, accessed: 2023-03-15
51. Verizon: 2024 data breach investigations report (2024), https://www.verizon.com/business/en-nl/resources/reports/2024/dbir/2024-dbir-data-breach-investigations-report.pdf, accessed: 2024-02-07
52. Wang, F., Ko, R., Mickens, J.: Riverbed: Enforcing user-defined privacy constraints in distributed web services. In: 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). pp. 615–630. USENIX Association, Boston, MA (Feb 2019), https://www.usenix.org/conference/nsdi19/presentation/wang-frank
53. Wang, L., Khan, U., Near, J., Pang, Q., Subramanian, J., Somani, N., Gao, P., Low, A., Song, D.: PrivGuard: Privacy regulation compliance made easier. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 3753–3770. USENIX Association, Boston, MA (Aug 2022), https://www.usenix.org/conference/usenixsecurity22/presentation/wang-lun
54. Xiao, Y., Li, Z., Qin, Y., Bai, X., Guan, J., Liao, X., Xing, L.: Lalaine: Measuring and characterizing Non-Compliance of apple privacy labels. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 1091–1108. USENIX Association, Anaheim, CA (Aug 2023), https://www.usenix.org/conference/usenixsecurity23/presentation/xiao-yue

A SEV-SNP

AMD deploys a separate processor, named AMD Security Processor (ASP), that creates a hardware root-of-trust and manages the security features of the TEE (e.g., signing the attestation report and preventing invalid accesses). Modifying guest register state and guest memory is possible only through the ASP, which tracks memory accesses and creates a hash of the initialization state. We use two TEE properties in Glasswall: (1) *initial state attestation* and (2) *information binding*, shown in Figure 6:

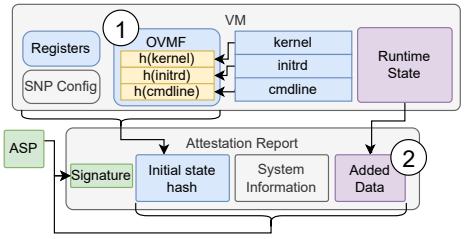


Fig. 6. SEV attestation mechanisms: An auditor verifies that the machine is started with a trusted initial image (1) that only loads safe code. The machine then can add information to the report at runtime (2), which the ASP signs. The public key of every chip is available from AMD.

(1) Initial state attestation. To secure a direct kernel boot, the hypervisor hashes the initial ramdisk (initrd), the kernel image, and the commandline, and stores the hashes into a defined table of a modified Open Virtual Machine Firmware (OVMF) [6,47]. The SEV state machine ensures that a VM can only be booted once the ASP has finalized the initial state hash. When booting, the OVMF bootloader calculates the hashes of the kernel-side elements and refuses to boot if they do not match. With these mechanics, the ASP only hashes the OVMF image, which includes hashes of the next stage (i.e., the direct kernel boot), up until the initial ramdisk. To enable attestation during runtime with fresh values, a secure system must request the ASP to sign additional data, which we discuss next.

(2) Information binding. The attestation report structure contains a field that allows supplying a small amount of data which will be signed by the ASP to exchange attested data. An attestation report by itself only guarantees that there was a VM created with this image *at some point*. To enable secure communication, a guest generates a public key pair and supplies their public key (or a hash thereof) to the ASP to request a signature binding the public key to this instance. The receiver verifies the attestation signature with the AMD-provided chip-specific public keys. It is paramount to understand that the channel is only secure if the image corresponding to the hash is secure, i.e., it does not provide access to the secret key to any other party.

B Protocol

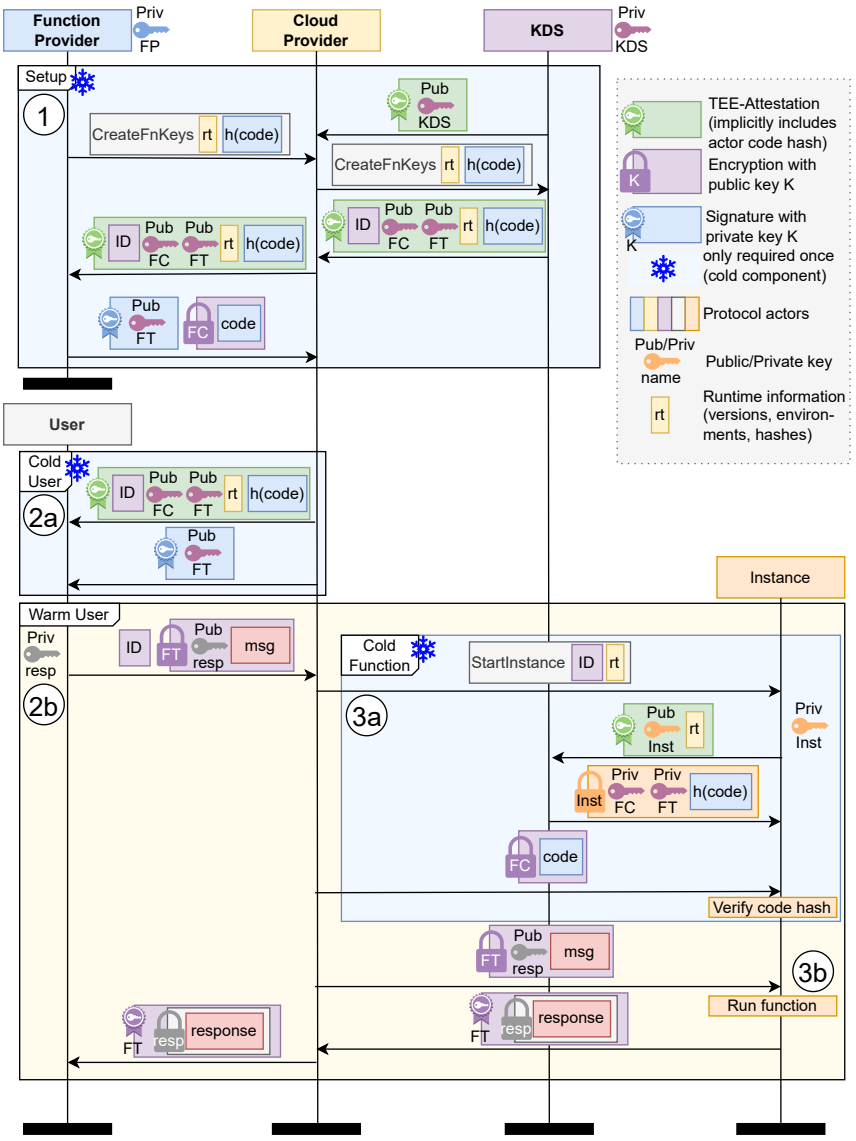


Fig. 7. Message Sequence Chart for the Glasswall Protocol.