A Verified Confidential Computing as a Service Framework for Privacy Preservation

Hongbo Chen[1*], Haobin Hiroki Chen[1], Mingshen Sun[2*], Kang Li[3*], Zhaofeng Chen[3*], and XiaoFeng Wang[1]

[1]*Indiana University Bloomington, {hc50,haobchen}@iu.edu, xw7@indiana.edu*
[2]*Independent Researcher, bob@mssun.me*
[3]*CertiK, {kang.li, zhaofeng.chen}@certik.com*

Abstract

As service providers are moving to the cloud, users are forced to provision sensitive data to the cloud. Confidential computing leverages hardware Trusted Execution Environment (TEE) to protect data in use, no longer requiring users' trust to the cloud. The emerging service model, Confidential Computing as a Service (CCaaS), is adopted by service providers to offer service similar to the Function-as-a-Serivce manner. However, privacy concerns are raised in CCaaS, especially in multi-user scenarios. CCaaS need to assure the data providers that the service does not leak their privacy to any unauthorized parties and clear their data after the service.

To address such privacy concerns with security guarantees, we first formally define the security objective, Proof of Being Forgotten (PoBF), and prove under which security constraints PoBF can be satisfied. Then, these constraints serve as guidelines in the implementation of the PoBF-compliant Framework (PoCF). PoCF consists of a generic library for different hardware TEEs, CCaaS prototype enclaves, and a verifier to prove PoBF-compliance. PoCF leverages Rust's robust type system and security features, to construct a verified state machine with privacy-preserving contracts. Last, the experiment results show that the protections introduced by PoCF incur minor runtime performance overhead.

1 Introduction

In recent years, cloud computing has gained significant momentum, revolutionizing the way individuals and organizations store, process, and access their data. However, amidst the growing popularity of cloud computing, a pressing concern arises: the protection of data privacy. In addition to algorithmic solutions (e.g., Multi-Party Computation [92]), confidential computing protects data in use through hardware-based trusted execution environments [26]. The development of hardware architectures, cloud infrastructures, and system-level middlewares has flourished in the past decades, making confidential computing (CC) more accessible through public clouds such as Google Cloud [43] and Microsoft Azure [63]. Both start-ups [35, 41] and established companies [19] offer solutions to businesses, such as healthcare, finance, blockchain, and privacy compliance.

Trusted Execution Environments (TEEs) have emerged as a solution for privacy concerns, providing a sheltered environment for computations and data analysis. TEEs ensure privacy by employing isolation, encryption, and attestation. The trusted processor creates a secure enclave, encrypting its memory and restricting access solely to the CPU. This protects the integrity of code and data within the enclave. With isolation and encryption combined, TEEs effectively maintain data confidentiality and privacy in computations. Moreover, TEEs support the attestation of the enclave program's authenticity to a remote verifier, providing the *cryptographic measurement* of the enclave and the *authenticity* of the processor to the verifier. allowing the data provider to confidently share sensitive information to the enclave, knowing that the *expected enclave program* is being executed by a genuine processor [22, 28].

Despite the sophisticated design of TEEs, privacy concerns continue to exist. While the measurement of the in-TEE program guarantees that the program is authentic, no evidence about its behavior is given. There is a risk that the enclave program may unintentionally leak sensitive data. Additionally, manually auditing the enclave program's source code to ensure the absence of data leaks is error-prone and time-consuming. Hence, there is a need to adopt additional security measures to enhance TEEs' privacy-preserving capabilities.

Recent advancements in frameworks have made Confidential Computing as a Service (CCaaS) increasingly available on cloud platforms [13, 72, 79, 84]. However, the exchange of information between the secure enclave and the untrusted environment raises privacy concerns, as private information may be leaked (e.g., via logging). The simplest solution to prevent such leaks would be to eliminate all communication between the enclave and the outside world, but this overkill completely eliminates usability and maintainability. Addi-

[*]Part of this work is done while the authors were at Baidu X-Lab.

tionally, privacy concerns extend beyond data leakage to the secret residue. To accelerate enclave initialization, CCaaS frameworks often serve multiple users after a single launch. Hence, if sensitive data remains in the enclave after one user's session, malicious attackers could steal it, e.g., by reading the heap where previous user's private data was not cleared [90].

Fortunately, while data providers must trust the hardware TEE, they can verify the enclave program instead of blindly relying on its measurement. Our goal is to address privacy leakage and residue threats in TEE applications and accommodate the privacy concerns of data providers by verifying the enclave program. Researchers have identified these problems and tried to solve them [4, 60, 90], but previous solutions face limitations as CC is moving to the cloud as a service. In more detail, they: 1) lack theoretic foundations; 2) do not handle leakage and residue problems together; 3) have a loss of generality for various hardware TEEs. Our solution aims to be more comprehensive, with formal foundations, while still being applicable to various hardware TEEs.

Our contributions. For confidential computing, we propose a privacy-preserving principle called *Proof of Being Forgotten* (PoBF). It requires that secrets are not leaked during the enclave execution and are zeroized immediately after use. PoBF regulates enclave program which deals with secrets (i.e., privacy) so that they are not revealed to any unauthorized party, and are consumed for processing or encrypted after use. As the essential property, Being Forgotten has two requirements for the enclave program: NoLeakage and NoResidue. NoLeakage requires that the enclave program cannot leak secrets during its execution; NoResidue requires that secrets are irrecoverably destructed or zeroized immediately after the execution. Note that "secrets" refers to both sensitive data provided by users and data tainted with sensitive data.

We also aim to provide a framework for building privacy-preserving enclave programs for confidential computing. The framework guarantees that the user's privacy is preserved in the course of the computation. Our effort starts with building a formal model of the generalized enclave and proving a series of theorems to establish the security constraints of PoBF. We then design and implement the *PoBF-Compliant Framework* (PoCF), which consists of three submodules: a PoBF-compliant library that is generic for different hardware TEEs, CCaaS prototypes in Intel SGX and AMD SEV, and a verifier to prove PoBF-compliance for the whole enclave program. When combined with verification technologies, the PoCF can be audited with confidence. Data providers can run the PoCF verifier on the enclave code, ensuring that the verified code is PoBF-compliant. Therefore, by confirming that the measurement of the remote enclave in remote attestation and the measurement derived from the locally built enclave are identical, the data provider knows that they meet the same PoBF property. Last, we evaluate our implementation and demonstrate that the protections in PoCF incur negligible overhead while meeting the security requirements.

In the design architecture, PoBF regulates privacy protections for data providers. On the implementation front, PoCF protects data confidentiality. Our code is available at [25]. In summary, we make the following contributions:

- Proposal of the PoBF principle with two requirements, leading to several formally proved security constraints.
- Design and implementation of PoCF, including a generic library for general CC tasks, a prototype enclave on Intel SGX, and a verifier to ensure PoBF compliance.
- Evaluation of the prototype CCaaS frameworks on security, overhead, real-world tasks, and its limitations.

Roadmap. We first introduce the background knowledge in § 2 and the detailed threat model in § 3. Secondly, the formalization of PoBF requirements and proof of the security constraints is presented in § 4. Guided by the constraints, we explain the design trade-offs and details of PoCF in § 5. Then, we exhibit the implementation in Rust and its verification towards PoBF properties in § 6. After that, PoCF is evaluated on security and performance in § 7. The development of verification tools for Rust is still at an early stage, so we discuss the limitations of our artifacts in § 8, especially on the modeling accuracy and precision in taint analysis. Finally, § 9 summarizes related work, and § 10 concludes our work.

2 Background

2.1 Confidential Computing

Hardware TEEs. Processor manufacturers and architecture designers have realized and released various designs of TEE, including AMD SEV, Intel SGX, and Intel TDX [28, 50, 56] for X86, TrustZone and CCA for ARM [8, 9], as well as various prototypes [37, 44, 59] for RISC-V. Among them, TDX, SEV, and CCA leverage the capabilities of virtual machines (VM) and can be regarded as VM-based TEEs. Intel SGX is a user-land TEE where the enclave program runs at the user level. All these TEEs separate the trusted world (or namely trusted *Realm* in ARM CCA and *Trusted Domain* in Intel TDX) from the normal (i.e., untrusted) world, and have instruction set architecture extensions (e.g., `EENTER` and `EEXIT` on Intel SGX) to control world switch. To defend against privileged attackers, e.g., system admin and operating system, a series of security primitives are recruited 1) isolated execution denies unauthorized access from privileged software; 2) remote attestation assures the end user that software running inside TEE is expected; 3) memory encryption mitigates hardware and software attacks related to the memory. Thus confidential computing is realized by leveraging hardware TEEs to protect data privacy during computing [26].

Confidential Computing as a Service. Based on hardware TEEs, the community offers several middlewares to support CC tasks. Some of these middlewares allow for CC tasks to

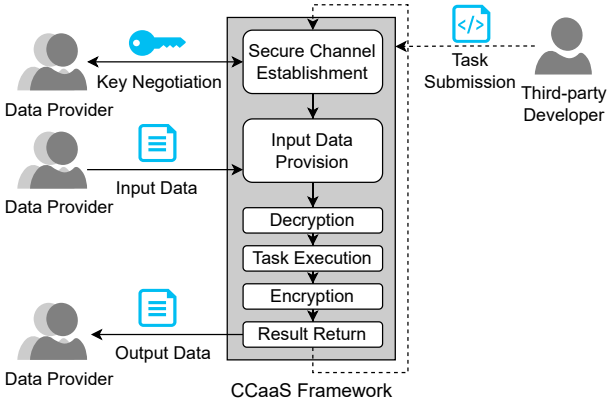


Figure 1: CCaaS workflow.

be performed in a similar manner to Function as a Service systems [72, 79, 84]. They typically provide pre-built tasks while also accepting code submissions from potentially malicious third-party vendors. The service starts after being initially deployed by system admins (e.g., on a TEE-enabled cloud). As depicted in Figure 1, the user first establishes a secure channel with the platform after remote attestation (e.g., RA-TLS [58]). Then the data provider provisions input data to the CCaaS framework, which later decrypts the input, executes the CC task, encrypts the result, and finally returns the output to the user. The CCaaS framework can be deployed as a service to support a multitude of data providers.

2.2 Static Program Analysis for Security

Rust Code. The Rust programming language embeds plenty of security checks (e.g., lifetime) within its compiler to achieve type safety, which implies memory safety [65, 67]. A resilient type system is instrumental in preventing type confusion that could lead to serious security concerns such as dereferencing invalid pointers and memory corruption [34]. Type-checking also empowers language-based access control.

Moreover, static code analyzers present additional means to bolster the security of Rust code. For example, *Prusti* is capable of verifying the functional behaviors of Rust programs using code annotations [12]. These annotations are converted automatically to formal proof with the help of an intermediate language, *Viper* [64]. Another example is *MIRAI*, an abstract interpreter of Rust's mid-level intermediate representation (MIR). It approximates the state of a program at any given point, allowing for the determination of certain properties statically [29, 30]. MIRAI also supports tag-based taint analysis which tracks the flow of data through a program [68].

Formal Method. Formal methods provide a way to specify a system and its properties using formal language. This allows for the properties to be expressed, proved, and mechanically checked, making the system's design theoretically secure. However, the complexity of programming languages used in production, particularly with the consideration of libraries, makes it hard to verify that the code satisfies the expected security properties. Developers may also introduce bugs that compromise the robustness of the program (e.g., using a cryptographic algorithm in the wrong way), deviating the whole system from the verified properties. Therefore, previous research turned to verify the methodologies [38, 52, 75].

3 Threat Model

First of all, the TEE hardware manufacturers such as Intel and AMD are trusted, so processors implementing TEEs and the software SDKs (usually offered by the manufacturers) running inside the secure enclaves are also considered trustworthy. Detailed threat models for different TEEs vary, but the privileged software and personnel outside the secure enclave are not trusted following the official threat models published by TEE manufacturers [10, 50, 51, 56]. For example, in the user-land TEE Intel SGX, guest and host OS, hypervisor, and BIOS are not trusted, whereas in the VM-based TEE (e.g., AMD SEV and Intel TDX), although the host OS is also untrusted, the guest VM itself is trusted. Nevertheless, these threat models share one common point: people with privileged access to the system (e.g., system administrator) are not trusted. Similar to previous work on building TEE containers [11, 23, 74], we believe side-channel attacks are orthogonal to our work and can be mitigated by other means, such as oblivious RAM [3, 5, 69]. Besides, denial of service is also not considered by PoBF and PoCF, and we discuss their effect and potential mitigation in § 8.

In CCaaS workflow, trust is a bit more intricate since enclave software can be developed by different software vendors: the framework may accept task code submitted by third-party developers. Our artifact, the PoCF Library & Enclave except for the CC task, is trusted but verified, as there may be potential security vulnerabilities (e.g., introduced by the developer). On the other hand, the CC task code uploaded by third-party developers is not trusted and must pass verification. This model is in line with the standard treatment of other CCaaS frameworks [72, 79]. When the CCaaS framework is hosted as a persistent enclave service to support multiple data providers, malicious data might be introduced into the system via their input. For example, the malicious input could be fed to exploit the platform, stealing the secrets of others who have been previously served by the framework [90]. So, PoCF does not trust the input from data providers.

From the stance of a data provider, the hardware TEE manufacturer, as well as the security primitives provided by it, are trusted. However, The cloud, its operators, and the software stack running outside the enclave are not trusted. The CCaaS framework running inside the enclave is trusted but verifiable, whereas the CC tasks submitted by third-party developers are untrusted and verified. Other data providers of the CCaaS are not trusted since their input may contain malicious exploits.

The CCaaS framework should protect her private data.

4 Formalizing PoBF

We formally define the concept of "Being Forgotten" and its two requirements for the purpose of establishing theoretic foundations to develop trustworthy enclave programs. However, these intuitive definitions alone have little practical meaning to the CC community. Developers need to know *how* to tweak their software to meet such requirements. Thus, we propose a generalized model for enclave programs that can be applied to various hardware TEEs. Based on the model, we showcase what enclave programs can *provably* satisfy PoBF requirements. The proof acts as security constraints for enclave design, implementation guidelines, and verification conditions for the PoCF Verifier.

4.1 Modeling the Enclave

General TEE Model with Tags. Processors with TEE support have two main architectural differences from those without. First, they introduce a new execution mode, namely `EnclaveMode` in our model, such that the program running in this mode can access the data belonging to the enclave. Second, they restrict access to the enclave from outside the enclave. We model the generalized TEE in Coq as shown in Table 1. The accessible locations from the viewpoint of programs can be either on the stack with an offset ($\texttt{Stack}(n)$), a place storing the return value (`RV`, e.g., `RAX` in x86-64 architecture), or an identifier of other places ($\texttt{Ident}(str)$, e.g., other general-purpose registers and heap locations denoted by an identifier str). Under this model, accessible memory could be regarded as a set of locations. While the cell c denotes storable locations from the architectural viewpoint. It contains a value v either in the untrusted world $\texttt{Normal}(v)$, the enclave $\texttt{Encalve}(et, v)$, or unused memory `Unused`. To secure sensitive data, we introduce two types of tags: 1) Security Tag vt, attached to the value v, to denote the security level; 2) Enclave Tag et to denote confined Zone locations inside the enclave. `Secret` should always be stored inside the confined `Zone`. Storable *me* can then be regarded as a mapping from locations l to cells c, encoded as a list of pairs $\texttt{List}(l, c)$. This mapping also provides information about how values are stored on the machine (e.g., in the `Zone` of the enclave). Reading a location involves iterating the storable *me* and obtaining a result r. If a location is present as a pair in *me*, r will be an $\texttt{Ok}(v)$, otherwise, an error $\texttt{Err}(e)$. Obviously, access to the enclave locations is restricted to `EnclaveMode`. Attempting to access them in `NormalMode` will result in $\texttt{Err}(\texttt{NoPrivilege})$, and attempting to access an unallocated location, `Unused`, will result in $\texttt{Err}(\texttt{Invalid})$. Finally, the state of the enclave st can be represented by a triplet $st = (me,\ mo,\ errs)$, where $errs$ denotes a list of errors that occurred if any. Raw pointers are not modeled to abstract away most memory errors as modern programming languages such as Java and Rust enforce type safety, eliminating memory errors. This aligns with our model and eliminates the need to model memory safety properties.

Table 1: Generalized model of secure enclaves.

Type	Sym.	Definition
Natural	n	$\in \mathbb{N}$
String	str	$\in \mathbb{S}$
Bool	b	$::= \texttt{True}\|\texttt{False}$
Value	v'	$::= \texttt{ConcreteN}(n)\|\texttt{ConcreteB}(b)\|\texttt{Any}$
Sec. Tag	vt	$::= \texttt{Secret}\|\texttt{NotSecret}\|\texttt{Nonsense}$
TagValue	v	$::= (v', vt)$
Mode	mo	$::= \texttt{EnclaveMode}\|\texttt{NormalMode}$
Location	l	$::= \texttt{Stack}(n)\|\texttt{Ident}(str)\|\texttt{RV}$
Enc. Tag	et	$::= \texttt{Zone}\|\texttt{NonZone}$
Cell	c	$::= \texttt{Nomral}(v)\|\texttt{Enclave}(et, v)\|\texttt{Unused}$
Result	r	$::= \texttt{Ok}(X)\|\texttt{Err}(e)$
Error	e	$::= \texttt{Invalid}\|\texttt{NoPrivilege}$
Storable	me	$::= \texttt{List}\ (l, c)$

Enclave Program Model. To model the program running inside the enclave, we utilized a modified `Imp` language to simplify programming language complexities, while retaining Turing completeness and expressiveness for reasoning TEE properties [70]. The syntax, as shown in Table 2, includes recursively defined *Expression* and *Procedure*. Expression includes a value (i.e., constant), accessing a location, as well as unary and binary operations. We extend the `Imp` language by adding two special statements (i.e., commands), named `Eenter` and `Eexit`, to switch into and out of `EnclaveMode`. Note here we deliberately use similar symbols of enclave enter and exit instructions (`EENTER` and `EEXIT`) in Intel SGX since they are semantically similar. Besides, procedures can contain commands of assignment and control flow statements, and their arguments are passed implicitly through storable locations. Procedures correspond to functions in real-world programming languages.

This syntax is capable of modeling the complex behaviors in mode switching and exceptions (i.e., errors and interrupts) for enclave programs. For example, Intel SGX handles Asynchronous Enclave Exits by saving the current execution context to the State Save Area and restoring the previous state saved at `EENTER` [28]. AMD SEV handles VMEXIT by saving the register state of the secure guest VM to the Virtual Machine Control Block [7]. These world switchings can be captured by `Eenter` and `Eexit` procedures, regardless of specific hardware TEE. State saving and restoring can be represented as a series of reads/writes to the storable using $\texttt{Asgn}\ l := e$.

4.2 PoBF Concepts

We outline the concepts of PoBF and their specific requirements in detail. First, we present a list of definitions, which are formally defined in our proof in Coq. We illustrate the

Table 2: Enclave program syntax.

Term	Sym.	Definition
Exp.	e	$::= l \mid v' \mid \texttt{UnaryOp}(e) \mid \texttt{BinaryOp}(e1, e2)$
Proc.	p	$::= \texttt{Nop} \mid \texttt{Eenter} \mid \texttt{Eexit} \mid \texttt{Asgn}\, l := e$ $\mid \texttt{If}\, e\, \texttt{Then}\, p1\, \texttt{Else}\, p2 \mid \texttt{While}\, e\, \texttt{Do}\, p$ $\mid p1;\, p2$

concepts in natural language here for better understanding.

Definition 4.1 (Critical State)**.** For a state $st = (mo, me, errs)$, if a cell c is in the Zone of the enclave memory and its value is tagged `Secret`, then the state st is critical.

Definition 4.2 (Leaked)**.** Given the storable me, if there exists a location l such that the corresponding cell c is not located in Zone and contains a value v tagged `Secret`, then predicate $\texttt{Leaked}(me)$ evaluates to `True`, otherwise `False`.

Definition 4.3 (Strong Residual)**.** Given the storable me, if there exists a location l such that the corresponding cell c stores a value v tagged `Secret`, then predicate strong $\texttt{Residual}'(me)$ evaluates to `True`, otherwise `False`.

Definition 4.4 (Weak Residual)**.** Weak `Residual` is the same as strong $\texttt{Residual}'$ except that it allows the return value location `RV` to hold a value v tagged `Secret`.

Definition 4.5 (Being Forgotten for A Procedure)**.** Being Forgotten $\texttt{BF}(p, st)$ is a predicate evaluating to `True` when `Leaked` evaluates to `False` all along the execution of p on initial state st, in conjunction with (weak or strong) `Residual` evaluates to `False` at ending state st' after p is executed. Otherwise, `BF` evaluates to `False`.

Here, `Leaked` and `Residue` are predicates, deriving two security requirements of `BF` – NoLeakage and NoResidue, each requiring that the corresponding predicates evaluate to `False`, i.e., $\texttt{BF} := \neg\texttt{Leaked} \wedge \neg\texttt{Residue}$. We expect the enclave program to have `BF` property, meaning that it meets both NoLeakage and NoResidue requirements. For NoLeakage requirement, if we want $\texttt{Leaked}(me)$ to evaluate to `False`, the secrets should remain within a designated "Zone" along the execution. However, for NoResidue, the strong $\texttt{Residual}'$ may lack practical meanings since it disallows inter-procedural secrets. So we define weak `Residual` where `Secret` value can be left on and passed via the return value `RV`. Therefore, the whole task satisfies `BF` when both `Leaked` and weak `Residue` evaluate to `False` for all the procedures in the task, except that strong $\texttt{Residual}'$ evaluate to `False` for the last procedure. Such treatment is practical and secure because it permits the secure flow of secrets and prevents leakage throughout the task. Being Forgotten for a task T, $\texttt{BF}(T)$, is defined in this way. We say a task T has the Proof of Being Forgotten (PoBF) property when $\texttt{BF}(T)$ evaluates to true, i.e., `BF` holds for task T.

4.3 PoBF Security Constraints

Single-user enclaves don't require handling of residue as they self-destruct after the execution. However, for multi-user enclaves serving persistently, privacy residue from users must be erased to prevent subsequent malicious attackers from learning sensitive information. We address the security constraints necessary to fulfill NoLeakage and NoResidue.

Theorem 4.1 (Constraints For NoLeakage)**.** For procedure p with initial state $st = (me,\ mo,\ errs)$, x'executing p does not leak the secret if: 1) the initial state st does not leak secret. 2) all memory writes ($\texttt{Asgn}\, l := e$) in p are within Zone if e is tagged with `Secret`. 3) p aborts when an error occurs.

This theorem is intuitive and mechanically proved in Coq. The first prerequisite establishes a clean starting point, which is satisfied right after enclave initialization. The second one mandates that all writes of `Secret` be confined within the Zone to prevent privacy breaches. The third constraint eliminates potential errors resulting from malfunctioning operations (e.g., accesses to `Unused` memory). Note that NoLeakage does not necessarily mean only these three conditions are met, as there may be alternative security constraints that provably guarantee NoLeakage. Prior to examining the theorem corresponding to NoResidue, an auxiliary procedure, *zeroize*, is introduced to aid in scrubbing privacy residue.

Definition 4.6 (Zeroize)**.** A procedure is defined as *zeroize* if, for the given storable $me = (l, c)$ denoting every location-cell pair in the Zone of the enclave, it clears the tagged values v stored in such cells c and sets the security tag to `NotSecret`.

Theorem 4.2 (Constraints for NoResidue)**.** For a procedure p, if it satisfies the NoLeakage requirement, the procedure p' derived by concatenating p and *zerorize* (i.e., $p' = p$; *zerorize*) satisfies the NoResidue requirement.

A challenge of applying Theorem 4.2 to every procedure is that all intermediate computation results must be cleared within the Zone, hindering the implementation. Alternatively, we only focus on procedures executed in critical states and loosen the requirement for *zeroize* to ignore `RV` for all procedures p. Procedures running in critical states must meet the NoResidue requirement. The last procedure of a task should not end in a critical state, which means that no secrete is presented (e.g., the final result is encrypted to `RV`). This can be achieved by instrumentation that executes a function *zerorize*() at the end of each procedure executed in critical states. Its implementation details are discussed in § 6.

Concurrent Execution. From a resource management standpoint, concurrent enclave programs can result in potential privacy leakage due to the sharing of resources among users. To mitigate this risk, shared resources should be disallowed. As a result, executing a multi-threaded enclave is equivalent to executing separate single-threaded tasks. A multi-threaded

enclave program without shared states satisfies the PoBF constraints, as each thread is a PoBF-compliant task.

4.4 PoBF Design Guidelines

The established security constraints discussed earlier can enlighten the design of PoBF-compliant enclave programs. We now provide a brief overview of how these theorems shape the development of enclave programs, and we present the detailed design and implementation of PoCF in § 5.

The concept of Zone can be approximated as a software sandbox where secret-related executions occur within it (e.g., memory reads and writes). Previous research has incorporated WebAssembly (WASM) into TEE, where WASM is interpreted in a strongly isolated sandbox [42, 61]. The Zone can also be viewed as the readable and writable locations for a function in the enclave program, including code, global variables, constants, input arguments, stack frame, and heap. Theorem 4.1 only imposes restrictions on writing `Secret`-tagged values, leading to two design options: simply confining writes of secrets to the Zone or conversely inspecting values written outside the Zone as `NotSecret`. Confining writes of secrets with a well-constructed data type can achieve NoLeakage, while software analysis tools can verify that values irrelevant to secrets are written outside the Zone. For secrets within the Zone, we can devise a comprehensive *zerorize* mechanism to erase all residual values regardless of security tags, including the stack frame, heap, and mutable global variables. Since the ciphertext is not secret, the last procedure in the task can encrypt the result and store it outside the Zone, and the encrypted result can be safely returned while the residual secrets are thoroughly erased. For multi-threaded enclave programs with no global variables and critical procedures that satisfy PoBF constraints, the program is PoBF-compliant.

5 PoBF-Compliant Framework

According to the formally proven security constraints, we design and implement a PoBF-Compliant Framework (PoCF). As illustrated in § 2, the framework can load CC Tasks submitted by developers, and allow users (i.e., data providers) to securely input their private data into the selected task within a secure enclave. Our design objectives, guided by the PoBF model, are separated into security requirements (SR) and auxiliary requirements (AR) as outlined below. It is important to note that all *plaintext* data from users is considered secret, while ciphertext cannot pose a threat to sensitive information under secure encryption so it is not secret.

SR1 Privacy leakage is statically detected for the enclave.

SR2 Secret residue is zeroized in all functions.

SR3 The CC Task meets the PoBF requirements (i.e., NoLeakage and NoResidue) and is verifiable to the framework deployer.

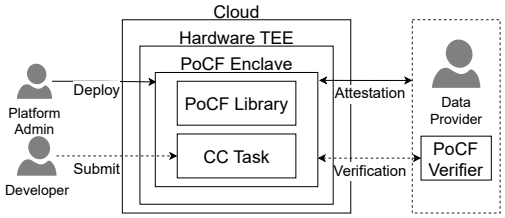


Figure 2: Architecture overview of PoCF.

SR4 PoBF compliance of the framework with CC Task is verifiable and attestable to users..

AR1 Minimal modifications for the CC Task code are required.

AR2 Different hardware TEEs are supported.

5.1 Overview

The design of PoCF and its interactions with stakeholders involved in CC are depicted in Figure 2. Specifically, PoCF is composed of four major components:

- *PoCF Library*: a hardware TEE-agnostic CCaaS library that integrates the core workflow as a state machine and high-level interfaces for confidential computing, as well as their annotated specifications.
- *CC Task*: a function conducting the actual confidential computing task that is invoked in a FaaS manner. It can be submitted by a third-party developer.
- *PoCF Encalve*: a hardware TEE-specific enclave program built on top of the PoCF Library with CC Task. It implements the interfaces specified by the library, such as remote attestation and encryption. It also manages the input from the untrusted source and output to the data provider. The enclave should satisfy NoLeakage and NoResidue requirements.
- *PoCF Verifier*: a tool that helps the platform deployers and data providers verify PoBF-compliance of the PoCF Enclave code, including CC Task and the PoCF Library.

Before the data provider initiates a task on the CCaaS framework, some preliminary steps must be taken. After the developer submits the CC Task, the platform administrator compiles the CC Task code along with the PoCF Enclave and deploys the service within the hardware TEE on the cloud. Since the platform deployer is not trusted, the data provider then verifies the (remote) enclave. Specifically, the source code of the the PoCF Enclave and CC Task is released and verified by the data provider using the PoCF Verifier, and the enclave measurement could be obtained by rebuilding the enclave binary (or image in VM-based TEE). After these preparatory steps, the data provider is convinced that remote enclaves with the calculated measurement cannot threaten her data privacy. In the service iteration, the data provider first conducts remote attestation and establishes a secure channel

with the enclave, confirming that the measurement of the service hosted on the cloud matches what has been verified and calculated by herself. The attestation report and verification result together serve as the "proof" to the data provider, guaranteeing that the enclave is not able to threaten her private data. Once trust is established between the data provider and the enclave service, the following steps in the CC workflow depicted in Figure 1 can be performed.

The design fulfills AR2 since the PoCF Library is platform-independent. We also show that little modification is needed for porting a CC Task into the enclave by an example illustrated in § 7.2. How SR1 and SR2 are satisfied are discussed respectively in the following subsections and proved in § 7.1. Nevertheless, they both rely on the integrity of the workflow, which is enforced by the PoCF Library and Rust. All these security building blocks are also verified by the PoCF Verifier, which meets SR3 and SR4. We defer the detailed discussions on verification to § 6.3 after presenting our implementation in detail. Besides, we also discuss how to leverage the confidential computing ecosystem to protect proprietary algorithms (i.e., verify the CC Task code without releasing it) in § 6.

5.2 The PoCF Library

The core library of PoCF is composed of three primary components: 1) a state machine that upholds CC workflow integrity, 2) concurrency support achieved by eradicating of shared states, and 3) a formal delineation of the state machine.

The State Machine and State Transition. The PoCF Library operates as the cornerstone for CCaaS frameworks across various TEEs. It fortifies CC workflow integrity and facilitates the enforcement of `Zerorize`. The library outlines a state machine that illustrates the workflow according to Figure 1. The detailed state transitions of CCaaS is depicted in Figure 3. Their informal and formal semantics are presented in Appendix A (see Table 6 and Figure 10). The key and data (abbreviated as K and D in the figure) also have associated states aligned with the task state, and their state transitions are also dictated within the library. Atop the state machine, the library designates a set of interfaces (i.e., state transition functions) and their contracts concerning the states (i.e., their expected behaviors and trait bounds). The interfaces are tailored to be implemented in TEE-specific enclaves, whereas the contracts are constraints that *must* be satisfied by the enclave. For instance, `establish_channel()` must be invoked in the `Initialized` task to establish a secure channel. The functions are realized in the PoCF Enclave, utilizing TEE-specific APIs (e.g., remote attestation). Nevertheless, we still need a formal specification that narrates the detailed semantics of the state machine, enabling its verification to ascertain correct implementations. This is critical because merely actualizing these interfaces does not assure their alignment with the specification, e.g., bugs could be inadvertently introduced by the developer.

Workflow Integrity. The PoCF Library must obstruct arbitrary invocation sequences of transition functions (e.g., returning results before encryption) by enforcing workflow integrity. Workflow integrity is based on but exceeds execution integrity. While execution integrity assures the soundness of the control flow graph (CFG), workflow integrity also stipulates the sequence of steps in CCaaS workflow (e.g., the result is initially encrypted, followed by its delivery to the data provider). If control flow integrity (CFI) is enforced, the execution integrity of the whole workflow can then be guaranteed by adhering to the contracts. Our abstract enclave model implicitly necessitates CFI and memory safety because it circumvents architectural details, particularly the behaviors of the pointer. Consequently, CFI is an implicit prerequisite for PoCF.

Concurrency. Our state machine fosters concurrency by deliberately circumventing shared state, a strategy inspired by systems such as Erlang and Medusa renowned for enabling concurrent execution [21, 85]. As a result, provided that the implementation ensures thread safety (e.g., through the language's runtime), a multi-threaded enclave service can operate concurrently without infringing upon data privacy.

5.3 Preventing Privacy Leakage

As highlighted in § 4.2, privacy leakage is considered as the disclosure of sensitive data to unconfined areas (i.e., data labeled as `Secret` being revealed in a non-Zone area). Such a scenario can arise due to a deliberate leak created by a malevolent developer or by capitalizing on the enclave's vulnerabilities. As a result, an attacker may partially or fully compromise data confidentiality. Therefore, in alignment with the design principles summarized in § 4, it is imperative that all the values being written to areas outside the Zone are not tagged `Secret`. Techniques such as information flow control [66] or taint analysis could be employed on enclave programs to guarantee that `Secret`-tagged variables are contained within the Zone. Potential leakage through edge functions, such as OCALL in Intel SGX, can be mitigated by such techniques. As edge functions serve as interfaces between the untrusted world and the enclave, so CC Tasks can embed the secret into the argument(s) of edge function calls to induce potential privacy leakage. To counter these security risks, we could either disallow all edge function calls in CC Tasks or verify that the arguments of these functions are not tainted by secrets. Additionally, the PoCF Enclave necessitates CFI and memory safety protections to prevent memory errors from being exploited to pilfer secrets, which are the same as the protections in the PoCF Library. These measures together fulfill SR1.

5.4 Scrubbing Secret Residue

The term "residue" pertains to changes on the storable (e.g., alterations of heap values) that occurs *after* a function has been

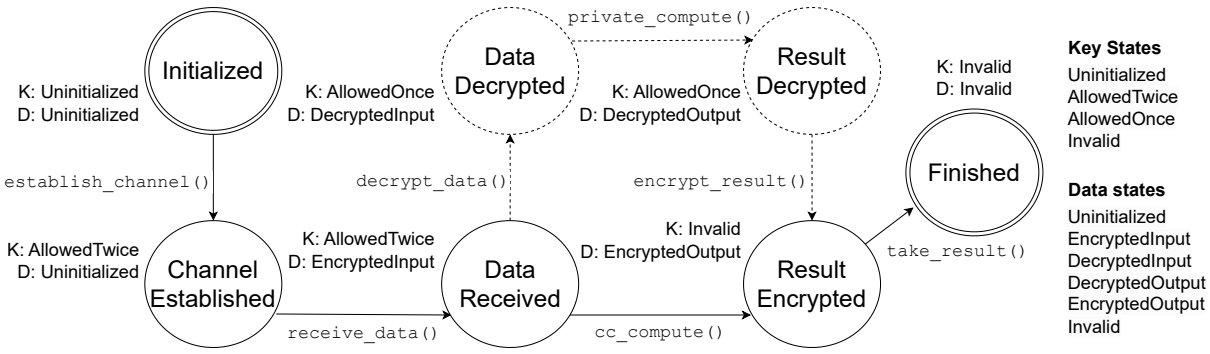


Figure 3: State transitions of PoCF task and associated key and data.

executed, where "storable" includes stack, heap, global variables, and external storage, which map to low-level hardware components such as memory, register, and disk. Neglected residue presents a risk. For example, inadequately cleared heaps used for storing plaintext human genomic data could allow attackers to purloin secrets by raw memory accesses. Nevertheless, for *ephemeral* enclaves which serve a single user in the CCaaS workflow and are used only once, the primary focus is on adhering to the NoLeakage requirement, as these enclaves are entirely dismantled after use. The state machine of the PoCF Library, in conjunction with hardware TEEs, ensures the destruction of the enclaves, thus thwarting any potential leaks of sensitive data via memory encryption. Even if the encrypted memory remains uncleared, the attacker cannot glean secret information from the ciphertext. Therefore, threats caused by residue are eliminated via the destruction of the ephemeral enclave, thus satisfying NoLeakage.

However, it is essential to note that while ephemeral enclave programs meet the NoResidue requirement, they do not feasibly support persistent services. This is because the enclave needs to be repeatedly initialized and destroyed for each new user request, resulting in substantial overhead.

Hence, when dealing with persistent enclaves serving multiple users, supplementary measures need to be enacted to tackle residue issues. As suggested in Theorem 4.2, one approach to mitigating residue involves incorporating a "zerorize" procedure immediately after critical procedures. By adapting the functions of enclave programs to clear sensitive data, the NoResidue requirement can be satisfied. Nevertheless, it is crucial to underscore that Theorem 4.2 assumes the eradication of potential leakage. Therefore, the scrubbing of privacy residue is only effective if the NoLeakage requirement is already satisfied.

For enclave programs, three aspects warrant consideration. First, the heap space allocated within or directly controlled by the function (e.g., reference to an object) that may store secrets, must be appropriately zeroized at the end of the function. Second, the function's stack frame could also contain secrets. This area typically remains uncleared before returning to the caller function and therefore, must be cleaned at the termination of the callee function. Finally, globally accessible locations (e.g., global variables) may also exist, However, as PoCF eschews these places, they do not demand our attention. Consequently, SR2 is satisfied by meticulously clearing the aforementioned areas.

6 Implementation and Verification of PoCF

This section delves into the implementation details of the PoCF Library and TEE-specific enclaves, along with their verification of PoBF-compliance. The PoCF Library is a general CCaaS workflow library compatible with various hardware TEEs. We also build the PoCF Enclaves upon the PoCF Library in SGX and SEV, representing the vastly deployed TEEs. The library and the enclave program are both written in Rust, a programming language that guarantees our implementation is free from memory errors. The PoCF Verifier is prototyped in Python, leveraging an abstract interpreter named MIRAI [36], the Prusti verifier, and the Rust compiler. The section also demonstrates the programming model and verification of the workflow from the viewpoint of the data provider.

6.1 TEE-agnostic Library

Implementation. According to the design, the library realizes a state machine and corresponding contracts. Benefiting from Rust's powerful type system, we encode a state machine with transition functions in typestate [77], where states are represented as generic parameters in pure Rust data structures (i.e., `struct`). With contracts (i.e., stipulations on the data structures) being described in `structs`' trait bounds, the required properties of the CCaaS workflow could be guaranteed by the type checker. Further, the Rust compiler carries out type checking statically, ensuring that the state machine and contracts are enforced with minimal runtime overhead.

Listing 1 showcases the code for `Task`, which incorporates the state machine as defined in Figure 3. `Task` has three generic parameters, `S`, `K`, and `D`, signifying the task's state, the key type, and the data type, respectively. This implementation not only models the state machine of the CCaaS workflow

but also the key and data tied to the task. The state of the key and data (`K` and `D`) are dependent on the state of the task `S`, so their transition possibilities are restricted accordingly. For example, when the task is in `ResultDecrypted` state, the key is `AllowedOnce`, meaning it can only be used once. Trait bounds serve as a contract for the generic types to implement the required interfaces. For instance, the key type must implement `Zerorize` which writes zeros on the memory housing it when it is `Invalid`ated. By implementing transition functions on the corresponding typestates with generics satisfying the contract, the state machine can be trustworthily constructed.

Listing 1: Typestate abstraction and specification.

```
1  pub struct Task<S, K, D> where
2      S: TaskState + DataState + KeyState,
3      K: Zeroize + Default, D: EncDec<K>,
4      <S as DataState>::State: DState,
5      <S as KeyState>::State: KState,
6  {
7      data: Data<<S as DataState>::State, D, K>,
8      key: Key<K, <S as KeyState>::State>,
9      _state: S,
10 }
11
12 pub trait TaskState {
13     #[pure]
14     fn is_initialized(&self) -> bool {false}
15     #[pure]
16     fn is_finished(&self) -> bool {false}
17     // Other similar functions are omitted.
18 }
19
20 pub struct Initialized;
21 #[refine_trait_spec]
22 impl TaskState for Initialized {
23     #[pure]
24     #[ensures(result == true)]
25     fn is_initialized(&self) -> bool {true}
26 }
27
28 #[ensures((&result)._state.is_allowed_once())]
29 // Other similar specifications are omitted
30 pub fn cc_compute(self) ->
31     Task<ResultEncrypted,Invalid,EncryptedOutput>;
```

Note that some states and transition functions in Figure 3 are represented by dotted lines, indicating they are private. These interfaces and typestates are exclusively visible to the PoCF Library but not to dependent artifacts *at the language level*. By capitalizing on this access control language feature, we can avert explicit access to sensitive data (e.g., `task.key`) and make private transitions atomic for the caller. Moreover, all fields in `Task` are all private, with the only visible function at `DataReceived` state being `cc_compute()`, which decrypts input data, executes the CC Task, and encrypts the outputs. This design obscures the states where the data is in plaintext, reducing the risk of data exposure. Additionally, as the `Task` struct does not include shared resources, it is self-contained and can be safely used in concurrent enclaves.

However, language-level access control alone is insufficient to ensure the privacy of secrets, especially in memory-unsafe languages like C. For instance, a direct dereference of a pointer to `Task` results in direct access to raw data. So, the TEE-specific enclave must mitigate such risks. We discuss the solution in the next subsection.

Scrubbing Residue. The transition functions correspond to the procedures in the PoBF formal model. The `Task` struct manages residue automatically, as its transition functions take itself as input and return a `Task` object in a new state, where *Zerorize()* is invoked as specified in the trait bounds. Although the state machine based on the typestate scrubs residue at the language level, sensitive data may linger in memory. The content of a freed (e.g., by `free()` in `libc`) memory block may still be partially recoverable by a raw pointer to that block, as `libc` does not specify freed memory blocks to be zeroed out. Therefore, we devise the `Zerorize` contract to handle residue. Transition functions are instrumented to enforce that stacks and registers except return values are cleared at the function epilogue. As for the heap, we patch the corresponding deallocator (e.g., `dealloc` function in Rust SGX SDK) to ensure that freed memories are zerorized. Besides, the protected `Key` struct must implement the `Zerorize` trait, as it must be invalidated before the task is `Finished`. This occurs when the output is encrypted using the key. The `Data` type does not need this contract since it can be encrypted in place, and the ciphertext is not secret. Thus, the weak `Residue` is evaluated as false in the transition functions.

Verification of the State Machine. Using the PoCF Library alone cannot fully satisfy the security requirements for enclave programs. To bridge the gap between the design and the implementation, we need a formal specification and mechanically check it in accordance with the code. This is achieved with *Prusti* [12], an automatic verification tool for Rust. We implement the typestate-based state machine as structs with associated traits (i.e., `TaskState`, `KState`, `DState` in Listing 1). To encode the specification, we write pure functions serving as the contract to represent its current state (line 13-16 in Listing 1). These functions aid Prusti in the verification of correctness, with which Prusti generate verification statements and proves them for each state.

6.2 The PoCF Enclave

Implementation. The PoCF Enclave is responsible for executing CC tasks securely. It is built on top of the PoBF Library with the CC Task code submitted by an untrusted developer. The platform oversees TEE-specific operations such as enclave initialization and remote attestation. The prototypes are implemented on Intel SGX AMD SEV due to their maturity. Adhering to the formal model that requires execution integrity and memory safety to be enforced, the SGX prototype utilizes Teaclave SGX SDK [86] and the SEV prototype deploys the standard library. The PoCF Enclave for SGX supports DCAP [47] and EPID [48] attestations,

and the SEV prototype uses Azure's attestation service [18]. These functions establish a secure channel after conducting remote attestation with the data provider. The SGX PoCF Enclave's implementation details will be mainly elaborated upon due to its unique programming models, and the corresponding SEV implementation removes the edge functions with OS. For SGX, only one ECALL is realized to avoid reentrant attacks [32, 57], while there are verified OCALLs for handling system functions with the untrusted OS. Although it's feasible to expose system interfaces directly to the CC Task, it could lead to extensive code modifications since the interfaces will be foreign functions that are challenging to use and verify directly. To resolve this, we encapsulate the system functions and macros with verification conditions. For instance in SGX, the `println!()` macro for logging in the CC task is wrapped using an OCALL, `printf()`, with a security level check predicate. These predicates are verified by MIRAI statically. System function parameters can contain secrets, and we now introduce our mitigation.

Preventing Leakage. Potential leakage locations are hardware TEE-specific. In SGX, data may be transmitted from the enclave to the untrusted world via edge calls (ECALLs and OCALLs) and memory access, such as explicit writes to memory outside the enclave. In SEV, network interfaces may leak secrets. Therefore, we could either prohibit these channels or impose restrictions (e.g., requiring encryption) on them to forestall leakage. We assume that the allocators, such as the allocator in `tlibc` in SGX, are safe (i.e., the allocated memory block is entirely within the enclave). Given that no pointers to the untrusted world can be legally initialized in the trusted world in safe Rust, the only potential sources of dangerous memory operations and pointers to the untrusted world are from edge functions. For example, they may return pointers to the untrusted world to the enclave. With only one ECALL in SGX that returns encrypted results and cannot leak sensitive data, we need to focus solely on edge functions, such as OCALLs in SGX and network interfaces in SEV.

Verification for Edge Function Calls. The PoCF Enclave code, including the CC Task, could trigger edge function calls. We stipulate that the parameters of the edge calls must be independent of secret and perform data flow tracking to validate such properties. In the CC workflow, excluding the CC Task where the code is untrusted, the tracking is carried out by the typestate. In the untrusted CC Task, all sensitive data structures such as `Key` and `Data` are tagged `Secret` when initialized, and we verify that edge call parameters do not contain the `Secret` tag. An abstract interpreter, `MIRAI`, is employed as the verifier. The tracking via MIRAI is based on the Rust Mid-level IR and could be imprecise. We conduct an evaluation on it in § 7.1. The `Secret` tag is propagated via various operations (e.g., slicing a part of the user data), and the tag is cleared upon the final encryption. Thus, MIRAI helps verify that edge calls leak no secret. Indeed, secret-dependent operations such as edge calls could potentially result in sensitive data leaks. However, this type of side/covert-channel is outside of our threat model, whereas we discuss tentative mitigations in § 8.

6.3 The PoCF Verifier

The data provider verifies the authenticity of the enclave software by rebuilding the enclave, acquiring its measurement, and conducting remote attestation, provided the source code is available. In cases where a proprietary algorithm is used in the CC Task and its source code is not public, a trusted build system can serve as a broker to verify the code and generate a trusted measurement [40]. The code is encrypted and supplied to the build system, which operates within a remotely attested hardware TEE. Data providers can delegate the verification job to the build system as it follows the steps to verify the PoCF Enclave. If verification is successful, the enclave binary/image and its measurement are produced. The developer and the data provider trust the build system after inspecting its source code, plus that attestation guarantees that the build system enclave is trusted, implying that the measurement of the PoCF Enclave built by this system is also authentic. Finally, when the data provider is about to transmit the secret to the PoCF Enclave, another remote attestation reassures the data provider that the remote enclave is the one previously built and verified by the trusted build system.

We now elaborate on how the verifier works. It leverages the Rust compiler, Prusti verifier, and MIRAI to accomplish the verification job. First, the Rust compiler checks memory and type safety. The PoCF Enclave forbids the use of `unsafe` code in its source code, except for specific places where encrypted input from the untrusted world is received and converted to vectors in safe Rust. These operations must be conducted through `unsafe` Rust but do not violate the PoBF property since the ciphertext is not secret. The prohibition of `unsafe` code prevents leakage via raw memory operations. The type checker, on the other hand, ensures that the state machine adheres to its contract. Second, Prusti guarantees the consistency of the typestate implementation with its specifications. In consequence, the NoResidue requirement is met since the `Zerorize` instrumentation can be executed in accordance with the state machine's specification. Finally, MIRAI performs taint analysis based on abstract interpretation. Each edge call is verified to ensure its parameters are not tagged `Secret`. The dependencies used by the PoCF Enclave invoking edge functions can also be verified. The PoCF Enclave only relies on the PoCF Library and a few third-party libraries, all of which contain no system interfaces [87]. The self-contained PoCF Library is written in pure Rust without external modules, containing no external function calls. Therefore, the dependencies of the PoCF Enclave thus require no further verification to satisfy NoLeakage.

Development under PoCF. From a CC Task programmer's

viewpoint, they can develop functions in FaaS as a normal Rust library. The only requirement is that the function must input and output serializable data types (e.g., `Vec<u8>`), where the data could be encrypted, decrypted, and transferred. The programmer can also run the PoCF Verifier on their end or delegate verification to the trusted builder before serving users.

7 Evaluation

In this section, we conduct security and performance evaluations of the PoCF implementations on Intel SGX and AMD SEV. On the one hand, we inspect our system on security and answer the question: *does PoCF reaches its goals of security*? On the other hand, we analyze the overhead of the PoCF Enclave and demonstrate its capability in real-world applications. We also make comparisons with related artifacts.

7.1 Security Analysis

We first outline the proof of PoCF satisfying the PoBF requirements using the PoCF semantics (see Table 6 and Figure 10 in Appendix). Then, we enumerate the threats that are mitigated by PoCF. Last, we evaluate the accuracy of dataflow tracking.

Theorem 7.1. The PoCF prototypes satisfy NoLeakage.

Proof Sketch. Following Theorem 4.1, we need to prove three constraints. Firstly, trivially, the initial state does not contain secrets. Secondly, the PoCF Verifier examines all edge function calls to guarantee that all stores of secrets cannot be passed via parameters. Also, in SGX, safe Rust guarantees that the enclave cannot write to untrusted memory, hence secret values cannot exit the enclave. Thirdly, PoCF aborts upon errors. Therefore, with the guarantee of control flow integrity in safe Rust, the PoCF prototypes cannot leak secrets. □

Theorem 7.2. The PoCF prototypes satisfy NoResidue.

Proof Sketch. To prove the theorem, we demonstrate that every function is correctly instrumented. All locations other than the return value that may store secrets should be properly zerorized. In safe Rust, residue can only persist on the heap, stack, register, and global variables. The PoCF Enclave has no global variables. The instrumented functions zerorizes the stack and register at the epilogue, which is enforced by the typestate. Finally, the modified allocator scrubs the memory used by the secrets on the heap when reclaiming objects. Therefore, PoCF does not have secret residue. □

Mitigated Threats. Table 3 summarizes the potential security threats that are mitigated by the PoCF framework and corresponding solutions. The threats are categorized into three classes accordingly: workflow integrity (WI), leakage (L), and residue (R). Various mitigation strategies are applied to handle these threats, and some mitigations are interleaved. For example, zerorize is based on the typestate which is verified according to formal specifications.

Table 3: Threats mitigated in PoCF prototypes.

Cat.	Description	Mitigated by
WI	Memory error in enclave	Safe Rust
WI	Control flow hijacking	Rust Compiler
WI	Workflow tampering	Typestate
WI	Race condition	Rust Compiler & Typestate
L	Leak via raw memory write	Safe Rust
L	Leak via edge calls	Taint Analysis
L	Unauthorized access	Typestate & Language Isolation
R	Residue in heap	Zerorize
R	Residue in stack/register	Instrumentation & Zerorize

Accuracy in Dataflow Tracking. Dataflow tracking in the state transitions, other than the CC Task, is accurate and secure as secrets are not accessible elsewhere in PoCF. The tracking here is performed and guaranteed by the type system. In the CC Task, while the key is invisible so its tracking is still accurate, the data can be referenced so we apply taint analysis to it. Therefore, we leverage MIRAI and analyze its accuracy via a series of tests covering features of Rust [36]. The evaluation results are summarized in Table 4.

MIRAI is accurate in modeling tag propagation at the computations, trait features, and in different control flows. However, it could fail when dealing with pointers and specific structs that need to propagate tags from their fields. This is attributed to the design flaw in MIRAI's heap model [1, 2], and we believe that adopting the methodology from [20] may enhance the accuracy. However, since safe Rust disallows raw pointer operations and PoCF only permits safe Rust in CC Task code, the PoCF Verifier works in most cases.

7.2 Performance Evaluation

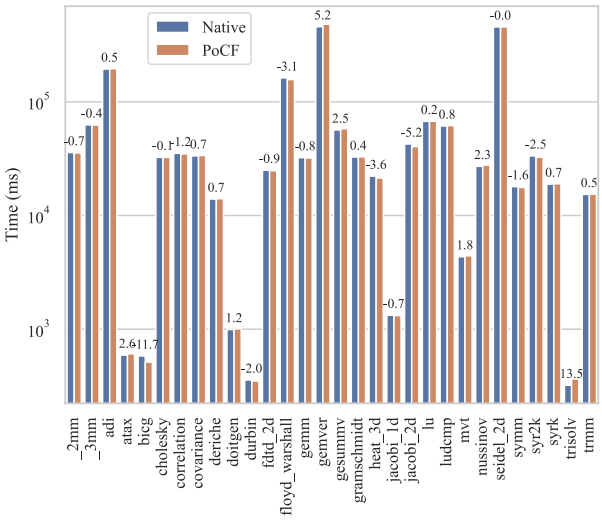
We conduct evaluations on both Intel SGX and AMD SEV platforms. The SGX server equips dual Xeon 5318S (24 cores each, hyperthreading enabled) CPU, 256GB RAM (64GB total EPC), and 2TB SSD. It runs Ubuntu 20.04.5 LTS with Linux 6.0, SGX SDK v2.17, and SGX DCAP v1.15. The SEV server is an Azure `DC16ads_v5` instance, which has 16 vCPUs, 64 GB RAM, and 256 GB SSD. It runs Ubuntu 22.04.2 LTS with Linux 5.15. To minimize the disturbance of the network, the data provider client program is deployed on the same machine as the PoCF Enclave.

Microbenchmarks. We unveil the performance overhead of PoCF protections via two microbenchmarks. An identity function is used as the CC Task to understand the performance of each component and of the whole workflow under different input sizes and various settings. We also evaluate the performance on a modified Rust implementation of Polybench [53, 71], a benchmark suite with computation-intensive tasks such as matrix multiplication. To weigh the effect of PoCF on computation, there is no input for Polybench eval-

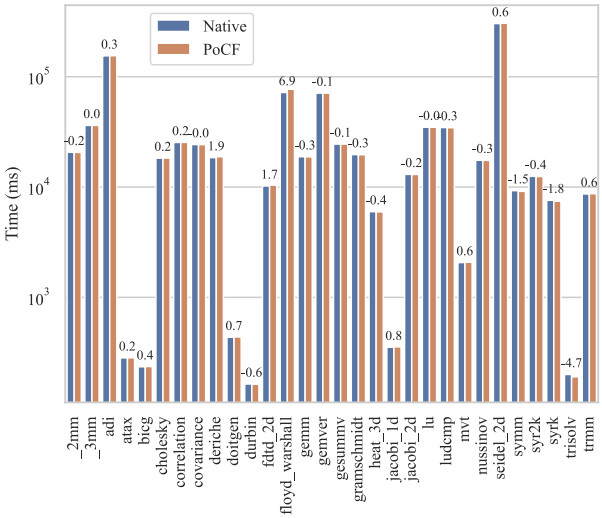
Table 4: The precision test of `MIRAI` categorized by Rust features.

Test Name	Covered Rust Features	Expected	Actual	Missed Feature(s)
`untrusted_input`	Traits, generics, and arrays	✓	✓	/
`control_flows`	Loops, branches, and pattern matches	✗: 1; ○: 5	○: 6	/
`ownership_transfer`	Ownership and borrow check	✗: 2	✗: 2	/
`pointers`	Smart and raw pointers	✗: 4	✗: 1	Leakage by `Rc<T>`, `Box<T>`, and `*const T`.
`complex_structs`	Collections and structs	✗: 4	✗:1	Tag propagation from field to the whole struct

All the tests were analyzed by `MIRAI` using its strictest analysis level, i.e., `MIRAI_FLAG=diag=paranoid`.
✓: No data leakage; ✗: Has data leakage; ○: Possible data leakage. The number behind "✗" or "○" denotes the number of data leakages.

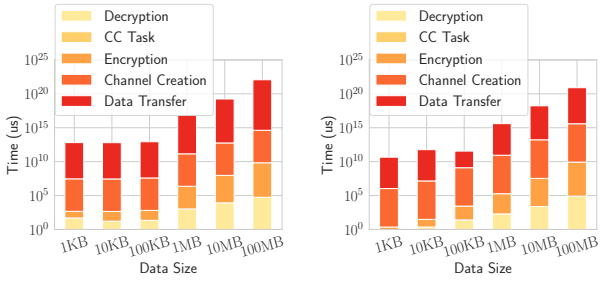


(a) Polybench: Performance of PoCF and Native on SGX.



(b) Polybench: Performance of PoCF and Native on SEV.

Figure 4: Performance of Polybench microbenchmarks.



(a) Cost breakup of PoCF on SGX. (b) Cost breakup of PoCF on SEV.

Figure 5: Identity task: Performance breakup of PoCF.

(a) Service time on SGX. (b) Service time on SEV.

Figure 6: Service time under the different cleared pages.

uations. The results can be found in Table 5, Figure 4, and Figure 5, with 10 repetitions of each subtask in each experiment setting. We measure the elapsed service time inside the enclave for the identity task and the elapsed time to execute the CC Task in Polybench. Native service runs inside an SGX enclave or encrypted VM in SEV without PoCF mitigations, while PoCF denotes our system in which `Zerorize` instrumentation is configured to clear at most 20 pages on the stack at function epilogues. The overhead is negligible, with average overheads in the identity task and Polybench being 1.24% and 0.28% respectively on SGX, and 0.86% and 0.94% on SEV. Such overheads imply PoCF protections cause a minor performance downgrade in computation. We also observe that the time spent on establishing a trusted channel is less relevant to the data size. However, the cost increases slowly with input sizes $<$ 10MB but linearly in 10-100MB, primarily due to the proportional growth of data-dependent workloads.

Overhead Analysis. Static checks (i.e., analysis of compiler

Table 5: Identity Task: Time (ms) under Different Data Sizes.

Config	1KB	10KB	100KB	1MB	10MB	100MB
NATIVE X	275.8	281.1	296.3	536.7	3026.5	28018.3
P W/O T X	278.3	280.4	298.6	541.1	3033.9	28022.9
P W/ T X	277.3	287.4	301.7	545.0	3043.7	28215.0
NATIVE V	489.1	487.3	449.7	495.6	502.0	923.3
POCF V	489.5	492.3	454.4	499.8	506.5	934.8

P: PoCF without data flow tracking; T data flow tracking; X: SGX; V: SEV

and MIRAI) incur no runtime overhead. Comparing the results of P W/O T X and P W/ T X in Table 5, we observe that typestate transition only incurs 0.82% performance degradation on average, which is very minor. Another portion of runtime overhead in PoCF comes from residue cleanup (i.e., zerorizing the stack and register), which incurs 0.42% overhead by comparing P W/O T X and NATIVE X. As shown in Figure 6, zerorizing more pages does not spend more time significantly, and we can even set it to the maximum stack size. In our evaluations, we set the number of cleared pages as 20 because it is enough for the benchmarks.

Macrobenchmarks. We port three the real-world applications to PoCF and other well-maintained SGX middlewares[1]. POCF and NATIVE are the same as those in the microbenmark. LINUX is the setting where NATIVE is executed in the normal world without using SGX. GRAMINE[2] is a C-based LibOS that supports unmodified binaries in SGX enclave [14]. OCCLUM is a Rust-based LibOS [74]. ENARX can host WebAssembly module for multiple TEE backends [13]. We use the release version of these middlewares: Gramine v1.3.1-1, Occlum 0.29.4-1, and Enarx 0.6.4. For all these evaluation settings other than ENARX and LINUX where the enclave size cannot be configured, we set the maximum enclave size to 8GB [3] and the maximum thread number to 16. We measure the elapsed time from the data provider's end since the LibOS is transparent to the process. All the steps in CCaaS are counted, except for LINUX where attestation cannot be performed. ENARX does not support multi-threading, so we omit its multi-threading performance.

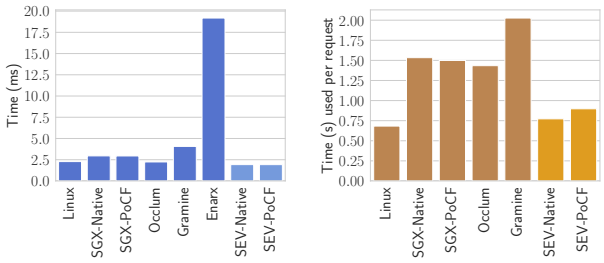
• *CPU-bound: AI Inference*. We use TVM to compile ResNet152 Model [17] as a library and link it to our framework, The Python script achieves this task in 125 lines of code (LoC), and the Rust function of the ResNet CC Task contains 50 LoC. We believe such porting effort is acceptable. It takes 600 KB `ndarray` input and outputs a label.

• *Memory-bound: FASTA*. We port the FASTA format genomic sequence generation algorithm [86]. This algorithm generates DNA sequences by copying from a given sequence and weighted random selection from two alphabets. The input and output sizes are both 4.4MB FASTA format sequences.

[1]We do not evaluate middlewares on SEV since it has no compatibility concern as SGX, hence few middlewares support SEV.

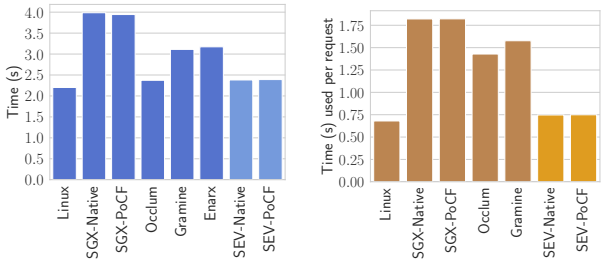
[2]Gramine is previously called Graphene [23].

[3]For Occlum, an additional 320MB memory is reserved for LibOS.



(a) Single-threaded. (b) Multi-threaded.

Figure 7: Macrobenchmark: AI inference execution time.

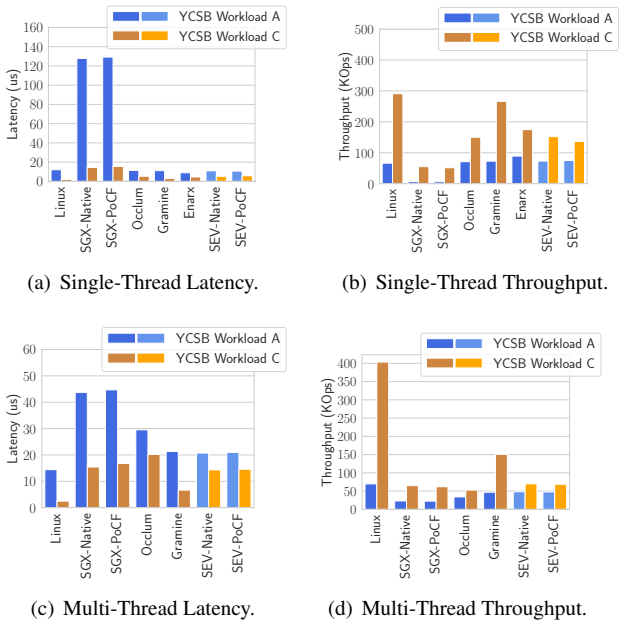


(a) Single-threaded. (b) Multi-threaded.

Figure 8: Macrobenchmark: FASTA execution time.

• *IO- and memory- bound: In-memory key-value database (KVDB)*. We use `hashbrown` [80], a Rust port of Google Swiss Table, to build an in-memory KVDB. We leverage YCSB [27], a standard benchmark for KV stores, to generate datasets and workloads in evaluations. The dataset contains 2^{20} KV of 8B keys paired with 1KB value, and the whole dataset is 1GB. We pick two workloads: A contains 50% reads and 50% writes, while C contains 100% reads. These two representative workloads are the lower and upper bounds for read-write ratios in YCSB, demonstrating a real-world deployment.

We depict the results of AI inference, FASTA, and KVDB in Figure 7, Figure 8 and Figure 9, respectively. In multi-threading scenarios of AI inference and FASTA tasks, we present the average service time per request after serving all four data providers. By comparing NATIVE with our solution, we confirm again that the overhead induced by PoCF protections is minor. In the KVDB payload where eight data providers dispatch queries, POCF incurs 1.12% and 6.92% overhead in single-thread mode and 1.98% and 4.95% overhead in multi-threading compared to NATIVE, respectively on SGX and SEV. In the AI inference task, we notice that PoCF has comparable performance with other TEE middlewares, both in single-threaded and multi-threaded scenarios (Figure 7). However, in the FASTA task, LibOSes-based solutions outperform PoCF, and their advantages are greater than those in the AI inference task. In KVDB payloads, LibOSes perform much better than SGX PoCF. We attribute this to data dependency and lack of I/O optimization in our SGX implementation, while SEV necessitates no network optimization.



(a) Single-Thread Latency. (b) Single-Thread Throughput.

(c) Multi-Thread Latency. (d) Multi-Thread Throughput.

Figure 9: Macrobenchmark: KVDB latency & throughput.

For example, Gramine and Occlum have rigorous optimizations on network I/O [15, 23]. We also observe that Occlum's performance downgrades in KVDB's multi-threading scenarios (Figure 9(c) and Figure 9(a)) [16]. Gramine and SEV experienced the same problem but to a less extent (Figure 9(d)). It may be caused by context switches, which offset the performance benefited from concurrency. However, this gap in single-threaded scenarios is significantly narrowed by concurrency. Our future work include I/O optimizations.

8 Discussion

Other Attack Vectors. Other attack vectors, such as side-channel attacks, although not within the scope of this work, can still lead to privacy breaches. However, different side-channels usually require orthogonal mitigations. For example, timing side-channels can be addressed by implementing constant-time operations [6]. MIRAI can also verify the constant-time property for the Rust code. For attacks on page [83, 89, 91] and micro-architectural side-channels [24, 73, 81, 82], separate low-level mechanisms can be utilized, such as binary rewriting [88] and instrumentation [49, 62]. Although these low-level mitigations can hardly be verified at the programming language level, the data provider can still verify that the remote enclave applies these defenses via remote attestation. Another potential threat is the denial of service (DoS) attack. If the enclave is suspended during the service (e.g., caused by an AEX without resuming in SGX), the residue may not be wiped out in this service cycle. However, DoS attacks cannot threaten privacy, as long as memory encryption is not compromised.

Rust Code Verfication. Verification on Rust code is still at an early stage. We tested many verification tools for Rust. Prusti strikes a balance between usability and complexity but can fail to verify some of the Rust features like nested generics and complex trait objects; the verification also takes a longer time. Rustbelt [54] reached its end of life in 2021 and no longer supports newer versions of Rust. Creusot [33], however, is strictly tied to a specific version of the nightly Rust toolchain that conflicts with Rust SGX SDK. Creusot's contracts are also intrusive, meaning that we need to write specifications for library functions. Aneaes [45] uses lambda calculus but is immature and does not support some common Rust features.

Another limitation is the imprecision of taint analysis tools. The inherent complexity plus the intricate design of Rust makes taint analysis awkward in dealing with some cases. For example, when a struct has tainted fields, it would not be propagated with a taint tag. Additionally, the approximation and modeling approaches adopted by taint analysis tools might be imprecise, resulting in issues such as over-tainting [55].

9 Related Work

Formal Models and Verification. Moat builds formal models for x86 with SGX instructions and the adversary [76]. It also builds a type system satisfying confidentiality. BesFS implements a series of filesystem interfaces for enclaves and proved its safety in Coq [75]. Subramanyan et al. also establish a model for secure remote execution of enclaves [78]. Komodo provides a verified software monitor implementing enclaves [39]. However, their code can hardly be utilized in real-world TEEs because of the lack of runtime support of the verification languages used to prove the properties. Besides, those models cannot be directly applied in CCaaS.

CCaaS Frameworks. In recent years, several frameworks has emerged to back CCaaS. Apache Teaclave, a FaaS platform, takes input from multiple parties to perform CC [79]. Enarx, Veracruz, and Oak integrate WebAssembly support to conduct confidential computing tasks [13, 72, 84]. There are also a lot of academia and industry projects making an effort to run unmodified binaries by offering a runtime [4, 23, 31, 46, 60, 74, 90]. However, these middlewares either bypass the privacy concerns or fail to provide a systematic solution to users.

10 Conclusion

This paper presents PoBF, a privacy protection principle for confidential computing, and PoCF, a PoBF-compliant Framework prototype. Leveraging Rust's safety features and type system, we design a state machine for CCaaS based on typestate, which supports different hardware TEEs. Besides, the PoCF verifier is realized to guarantee two PoBF requirements

are satisfied. The evaluations show that PoCF protections incur minor runtime overhead to achieve security goals.

Acknowledgements

We sincerely thank our shepherd and the anonymous reviewers for their valuable feedback, and Weijie Liu for the technical discussions. Authors from Indiana University Bloomington are supported in part by NSF-1838083, 2207231, and NIH R01HG010798.

References

[1] Design documentation of MIRAI. https://github.com/facebookexperimental/MIRAI/blob/41b3c946163df3faf543667b64448bd2abc3357f/documentation/TagAnalysis.md, 2022.

[2] How to verify tag when type changes? https://github.com/facebookexperimental/MIRAI/issues/1131#issuecomment-1039643992, 2022.

[3] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. Obfuscuro: A commodity obfuscation engine on intel sgx. In *Network and Distributed System Security Symposium*, 2019.

[4] Adil Ahmad, Juhee Kim, Jaebaek Seo, Insik Shin, Pedro Fonseca, and Byoungyoung Lee. Chancel: efficient multi-client isolation under adversarial programs. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2021.

[5] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. Obliviate: A data oblivious filesystem for intel sgx. In *NDSS*, 2018.

[6] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *USENIX Security Symposium*, volume 16, pages 53–70, 2016.

[7] AMD. Secure encrypted virtualization api version 0.24. https://www.amd.com/system/files/TechDocs/55766_SEV-KM_API_Specification.pdf, Apr. 2020.

[8] ARM. Arm security technology building a secure system using trustzone technology. https://developer.arm.com/documentation/PRD29-GENC-009492/c?lang=en, Apr. 2009.

[9] ARM. Arm cca security model 1.0. https://developer.arm.com/documentation/DEN0096/latest, Feb. 2021.

[10] ARM. Arm confidential compute architecture software stack. https://developer.arm.com/documentation/den0127/0100/, Sep. 2021.

[11] 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. SCONE: Secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[12] Vytautas Astrauskas, Aurel Bílỳ, Jonáš Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J Summers. The prusti project: Formal verification for rust. In *NASA Formal Methods: 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24–27, 2022, Proceedings*, pages 88–108. Springer, 2022.

[13] The Enarx Authors. The enarx project. https://github.com/enarx/enarx, 2022.

[14] The Gramine Authors. The gramine project. https://gramineproject.io, 2022.

[15] The Occlum Authors. Optimize the perf of sendmsg/recvmsg by allocating untrusted buffers. https://github.com/occlum/occlum/commit/e352a190ea63cc9370371026e9bf620f92b24b41, 2020.

[16] The Occlum Authors. Network performance tuning record. https://github.com/occlum/occlum/issues/1092, 2022.

[17] The ONNX Authors. Resnet152 model in the form of onnx. https://github.com/onnx/models/blob/main/vision/classification/resnet/model/resnet152-v1-7.onnx, 2020.

[18] Microsoft Azure. Attestation services | microsoft azure. https://azure.microsoft.com/en-us/products/azure-attestation.

[19] Springer security and policy compliance platform - baidu ai cloud. https://cloud.baidu.com/product/springer.html, 2022.

[20] Abhiram Balasubramanian, Marek S Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. System programming in rust: Beyond safety. In *Proceedings of the 16th workshop on hot topics in operating systems*, pages 156–161, 2017.

[21] Thomas W Barr and Scott Rixner. Medusa: Managing concurrency and communication in embedded systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 439–450, 2014.

[22] Robert Buhren, Christian Werling, and Jean-Pierre Seifert. Insecure until proven updated: analyzing amd sev's remote attestation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1087–1099, 2019.

[23] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658. USENIX Association, 2017.

[24] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 142–157. IEEE, 2019.

[25] Hongbo Chen, Hiroki H Chen, and Mingshen Sun. ya0guang/pobf: Proof of being fogotten (pobf) and pobf-compliant framework (pocf) prototypes. https://github.com/ya0guang/PoBF, 2023.

[26] Confidential Computing Consortium. Confidential computing: Hardware-based trusted execution for applications and data. Technical report, Confidential Computing Consortium, jan 2021.

[27] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[28] Victor Costan and Srinivas Devadas. Intel sgx explained. *Cryptology ePrint Archive*, 2016.

[29] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.

[30] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282, 1979.

[31] Jinhua Cui, Shweta Shinde, Satyaki Sen, Prateek Saxena, and Pinghai Yuan. Dynamic binary translation for sgx enclaves. *ACM Transactions on Privacy and Security*, 25(4):1–40, 2022.

[32] Jinhua Cui, Jason Zhijingcheng Yu, Shweta Shinde, Prateek Saxena, and Zhiping Cai. Smashex: Smashing sgx enclaves using exceptions. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 779–793, 2021.

[33] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. Creusot: a foundry for the deductive verification of rust programs. In *Formal Methods and Software Engineering: 23rd International Conference on Formal Engineering Methods, ICFEM 2022, Madrid, Spain, October 24–27, 2022, Proceedings*, pages 90–105. Springer, 2022.

[34] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. Memory safety without runtime checks or garbage collection. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, LCTES '03, page 69–80, New York, NY, USA, 2003. Association for Computing Machinery.

[35] Edgeless systems — confidential computing at scale for everyone. https://www.edgeless.systems, 2022.

[36] Facebook. Mirai: an abstract interpreter for the rust compiler's mid-level intermediate representation. https://github.com/facebookexperimental/MIRAI, 2022.

[37] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Scalable memory protection in the PENGLAI enclave. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, 2021.

[38] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 287–305, 2017.

[39] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 287–305, 2017.

[40] Andrew Ferraiuolo, Razieh Behjati, Tiziano Santoro, and Ben Laurie. Policy transparency: Authorization logic meets general transparency to prove software supply chain integrity. 2022.

[41] Fortanix - data-first multicloud security, 2022.

[42] David Goltzsche, Manuel Nieke, Thomas Knauth, and Rüdiger Kapitza. Acctee: A webassembly-based two-way sandbox for trusted resource accounting. In *Proceedings of the 20th International Middleware Conference*, pages 123–135, 2019.

[43] Confidential computing — google cloud. https://cloud.google.com/confidential-computing/, 2022.

[44] HEX-Five. Multizone risc-v datasheet. Technical report, HEX-Five, jan 2020.

[45] Son Ho and Jonathan Protzenko. Aeneas: Rust verification by functional translation. *Proceedings of the ACM on Programming Languages*, 6(ICFP):711–741, 2022.

[46] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. *ACM Transactions on Computer Systems (TOCS)*, 35(4):1–32, 2018.

[47] Intel. Attestation for data center orientation guide. https://www.intel.com/content/dam/develop/public/us/en/documents/intel-sgx-dcap-ecdsa-orientation.pdf.

[48] Intel. A cost-effective foundation for end-to-end iot security. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-epid-white-paper.pdf.

[49] Intel. Intel software guard extensions (sgx) sw development guidance for potential bounds check bypass (cve-2017-5753) side channel explots. https://www.intel.com/content/dam/develop/external/us/en/documents/180204-sgx-sdk-developer-guidance-v1-0.pdf.

[50] Intel. Intel® trust domain extensions (intel® tdx). https://cdrdv2.intel.com/v1/dl/getContent/690419, Aug. 2021.

[51] Intel. Product brief: Intel® software guard extensions. https://cdrdv2.intel.com/v1/dl/getContent/723693?explicitVersion=true, Mar. 2022.

[52] Evan Johnson, David Thien, Yousef Alhessi, Shravan Narayan, Fraser Brown, Sorin Lerner, Tyler McMullen, Stefan Savage, and Deian Stefan. Доверяй, но проверяй: Sfi safety for native-compiled wasm. In *Network and Distributed Systems Security (NDSS) Symposium*, 2021.

[53] JRF63. polybench-rs. https://github.com/JRF63/polybench-rs, 2020.

[54] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34, 2017.

[55] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. Dta++: dynamic taint analysis with targeted control-flow propagation. In *NDSS*, 2011.

[56] David Kaplan, Jeremy Powell, and Tom Woller. Amd sev-snp: Strengthening vm isolationwith integrity protection and more. Technical report, Technical Report. Advanced Micro Devices Inc., 2020.

[57] Mustakimur Rahman Khandaker, Yueqiang Cheng, Zhi Wang, and Tao Wei. Coin attacks: On insecurity of enclave untrusted interfaces in sgx. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 971–985, 2020.

[58] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. Integrating remote attestation with transport layer security. *arXiv preprint arXiv:1801.05863*, 2018.

[59] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Dawn Song, and Krste Asanović. Keystone: An open framework for architecting tees. *arXiv preprint arXiv:1907.10119*, 2019.

[60] Weijie Liu, Wenhao Wang, Hongbo Chen, XiaoFeng Wang, Yaosong Lu, Kai Chen, Xinyu Wang, Qintao Shen, Yi Chen, and Haixu Tang. Practical and efficient in-enclave verification of privacy compliance. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 413–425. IEEE, 2021.

[61] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. Twine: An embedded trusted runtime for webassembly. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 205–216. IEEE, 2021.

[62] Microsoft. Qspectre. https://learn.microsoft.com/en-us/cpp/build/reference/qspectre?view=msvc-170.

[63] Introducing azure confidential computing. https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing, 2022.

[64] Peter Müller, Malte Schwerhoff, and Alexander J Summers. Viper: A verification infrastructure for permission-based reasoning. In *International conference on verification, model checking, and abstract interpretation*, pages 41–62. Springer, 2016.

[65] Andrew Myers. Tutorial t1: Expressing and enforcing security with programming languages, June 2006.

[66] Andrew C Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, 1999.

[67] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):477–526, 2005.

[68] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, volume 5, pages 3–4. Citeseer, 2005.

[69] Hyunyoung Oh, Adil Ahmad, Seonghyun Park, Byoungyoung Lee, and Yunheung Paek. Trustore: Side-channel resistant storage for sgx using intel hybrid cpu-fpga. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1903–1918, 2020.

[70] Benjamin C Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Catalin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. Logical foundations. *Software Foundations series*, 1, 2018.

[71] Louis-Noël Pouchet et al. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 437:1–1, 2012.

[72] project-oak/oak. https://github.com/project-oak/oak, 2022.

[73] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1852–1867. IEEE, 2021.

[74] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and efficient multitasking inside a single enclave of intel sgx. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 955–970, 2020.

[75] Shweta Shinde, Shengyi Wang, Pinghai Yuan, Aquinas Hobor, Abhik Roychoudhury, and Prateek Saxena. Besfs: A posix filesystem for enclaves with a mechanized safety proof. In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 523–540, 2020.

[76] Rohit Sinha, Sriram Rajamani, Sanjit Seshia, and Kapil Vaswani. Moat: Verifying confidentiality of enclave programs. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 1169–1184, New York, NY, USA, 2015. Association for Computing Machinery.

[77] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, SE-12(1):157–171, 1986.

[78] Pramod Subramanyan, Rohit Sinha, Ilia Lebedev, Srinivas Devadas, and Sanjit A Seshia. A formal foundation for secure remote execution of enclaves. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 2435–2450, 2017.

[79] Apache teaclave (incubating). https://teaclave.apache.org, 2022.

[80] The Rust-Lang Team. Rust port of google's high-performance swiss-stable hash map. https://github.com/rust-lang/hashbrown, 2023.

[81] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *Proceedings fo the 27th USENIX Security Symposium*. USENIX Association, 2018.

[82] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 54–72. IEEE, 2020.

[83] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *Proceedings of the 26th USENIX Security Symposium*, pages 1041–1056. USENIX Association, 2017.

[84] Veracruz project – just another linux foundation projects 2 site. https://veracruz-project.com, 2022.

[85] Steve Vinoski. Concurrency with erlang. *IEEE Internet Computing*, 11(5):90–93, 2007.

[86] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. Towards memory safe enclave programming with rust-sgx. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2333–2350, 2019.

[87] Pei Wang, Yu Ding, Mingshen Sun, Huibo Wang, Tongxin Li, Rundong Zhou, Zhaofeng Chen, and Yiming Jing. Building and maintaining a third-party library supply chain for productive and secure sgx enclave development. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, pages 100–109, 2020.

[88] Shuai Wang, Wenhao Wang, Qinkun Bao, Pei Wang, XiaoFeng Wang, and Dinghao Wu. Binary code retrofitting and hardening using sgx. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*, pages 43–49, 2017.

[89] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2421–2434, 2017.

[90] Wenhao Wang, Weijie Liu, Hongbo Chen, Xiaofeng Wang, Hongliang Tian, and Dongdai Lin. Trust beyond border: Lightweight, verifiable user isolation for protecting in-enclave services. *IEEE Transactions on Dependable and Secure Computing*, 2021.

[91] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656. IEEE, 2015.

[92] Andrew C Yao. Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*, pages 160–164. IEEE, 1982.

A Semantics of PoCF

We use $e \Downarrow^{st}_{st'} r$ to denote that an expression e starting with a given state $st = (me, mo, errs)$, evaluates to r while resulting in an updated state $st' = (me', mo', errs)$. The typestate σ is a triplet $\sigma := (\mathsf{TaskState}, \mathsf{KeyState}, \mathsf{DataState})$ where TaskState denotes the state of the `Task` struct, KeyState denotes the state of the session key, and DataState denotes the state of the confidential data uploaded by the data provider. The key k and the data d are tuples of tagged values $v := (v', vt)$. We use the notation $\pi := (k, d, \sigma)$ to denote the input and output (wrapped in the result r) of the transition functions in the PoCF workflow. That is, for a given transition function f, we have $f(\pi) \Rightarrow r(\pi')$, where "$\Rightarrow$" means "evaluates to". The description of the transition functions in the PoCF workflow can be found in Table 6 and Figure 3, and the formal semantics thereof is described in Figure 10.

Table 6: Informal semantics of PoCF state transitions.

Transition Function	Semantics	Operations and Side Effect
`establish_channel`(f : `Fn`() → `Key`)	Do remote attestation and get the session key	`task.key` = f()
`receive_data`(f : `Fn`() → `Data`)	Receive encrypted data from an untrusted channel	`task.data` = f()
`decrypt_data`()	Decrypt the data using the session key	`task.data` = `task.data.decrypt(task.key)`
`do_compute`(f : `Fn(Data)` → `Data`)	Conduct CC task computation on the decrypted data	`task.data` = f(`task.data`)
`encrypt_result`()	Encrypt the data using the session key and drop the key	`task.data` = `task.data.encrypt(task.key)`
`take_result`() → `Data`	Take the encrypted result and destruct the task	return `task.data` and drop `task`

Note that every transition function also takes the `Task` (i.e., `self` in Rust) and returns a `Task`, except for `take_result()`.

$$\frac{\texttt{Leaked}(st) \Rightarrow \texttt{false} \quad f : () \to str \quad k = (\texttt{Any}, \texttt{Nonsense}) \quad d = (\texttt{Any}, \texttt{Nonsense}) \quad \sigma = (\mathsf{Initialized}, \mathsf{Uninitialized}, \mathsf{Uninitialized}) \quad \pi = (k, d, \sigma) \quad \{\texttt{establish\_channel}(\pi, f); \mathit{zerorize}(me')\} \Downarrow^{st}_{st'} r}{r = \texttt{Ok}(\pi') \vdash \sigma' = (\mathsf{ChannelEstablished}, \mathsf{AllowedTwice}, \mathsf{Uninitialized}), k' = (str, \texttt{Secret}), d' = (\texttt{Any}, \texttt{Nonsense})}$$

$$\frac{\texttt{Leaked}(st) \Rightarrow \texttt{false} \quad f : () \to str \quad k = (str, \texttt{Secret}) \quad d = (\texttt{Any}, \texttt{Nonsense}) \quad \sigma = (\mathsf{ChannelEstablished}, \mathsf{AllowedTwice}, \mathsf{Uninitialized}) \quad \pi = (k, d, \sigma) \quad \{\texttt{receive\_data}(\pi, f); \mathit{zerorize}(me')\} \Downarrow^{st}_{st'} r}{r = \texttt{Ok}(\pi') \vdash \sigma' = (\mathsf{DataReceived}, \mathsf{AllowedTwice}, \mathsf{EncryptedInput}), k' = (str, \texttt{Secret}), d' = (str, \texttt{NotSecret})}$$

$$\frac{\texttt{Leaked}(st) \Rightarrow \texttt{false} \quad k = (str, \texttt{Secret}) \quad d = (str, \texttt{NotSecret}) \quad \sigma = (\mathsf{DataReceived}, \mathsf{AllowedTwice}, \mathsf{EncryptedInput}) \quad \pi = (k, d, \sigma) \quad \{\texttt{decrypt\_data}(\pi, \bot); \mathit{zerorize}(me')\} \Downarrow^{st}_{st'} r}{r = \texttt{Ok}(\pi') \vdash \sigma' = (\mathsf{DataDecrypted}, \mathsf{AllowedOnce}, \mathsf{DecryptedInput}), k' = (str, \texttt{Secret}), d' = (str, \texttt{Secret})}$$

$$\frac{\texttt{Leaked}(st) \Rightarrow \texttt{false} \quad f : str \to str \quad k = (str, \texttt{Secret}) \quad d = (str, \texttt{Secret}) \quad \sigma = (\mathsf{DataDecrypted}, \mathsf{AllowedOnce}, \mathsf{DecryptedInput}) \quad \pi = (k, d, \sigma) \quad \{\texttt{private\_compute}(\pi, f); \mathit{zerorize}(me')\} \Downarrow^{st}_{st'} r}{r = \texttt{Ok}(\pi') \vdash \sigma' = (\mathsf{ResultDecrypted}, \mathsf{AllowedOnce}, \mathsf{DecryptedOutput}), k' = (str, \texttt{Secret}), d' = (str, \texttt{Secret})}$$

$$\frac{\texttt{Leaked}(st) \Rightarrow \texttt{false} \quad k = (str, \texttt{Secret}) \quad d = (str, \texttt{Secret}) \quad \sigma = (\mathsf{DataDecrypted}, \mathsf{AllowedOnce}, \mathsf{DecryptedOutput}) \quad \pi = (k, d, \sigma) \quad \{\texttt{encrypt\_result}(\pi, \bot); \mathit{zerorize}(me')\} \Downarrow^{st}_{st'} r}{r = \texttt{Ok}(\pi) \vdash \sigma' = (\mathsf{ResultEncrypted}, \mathsf{Invalid}, \mathsf{EncryptedOutput}), k' = (str, \texttt{NotSecret}), d' = (str, \texttt{NotSecret})}$$

$$\frac{\texttt{Leaked}(st) \Rightarrow \texttt{false} \quad k = (str, \texttt{NotSecret}) \quad d = (str, \texttt{NotSecret}) \quad \sigma = (\mathsf{DataEncrypted}, \mathsf{Invalid}, \mathsf{EncryptedOutput}) \quad \pi = (k, d, \sigma) \quad \{\texttt{take\_result}(\pi, \bot); \mathit{zerorize}(me')\} \Downarrow^{st}_{st'} r}{r = \texttt{Ok}(\pi) \vdash \sigma' = (\mathsf{Finished}, \mathsf{Invalid}, \mathsf{Invalid}), k' = (\texttt{Any}, \texttt{Nonsense}), d' = (\texttt{Any}, \texttt{Nonsense})}$$

Figure 10: Formal semantics of PoCF workflow