A Verified Confidential Computing as a Service Framework for Privacy Preservation

Hongbo Chen\textsuperscript{1}, Haobin Hiroki Chen\textsuperscript{1}, Mingshen Sun\textsuperscript{2}, Kang Li\textsuperscript{3}, Zhaofeng Chen\textsuperscript{3}, and XiaoFeng Wang\textsuperscript{1}

\textsuperscript{1} Indiana University Bloomington, \textsuperscript{2} Independent Researcher, \textsuperscript{3} CertiK
Introduction & Background
Coffee Incidents
Privacy Incidents
TEE’s Abilities and Inabilities

- Attestation: Guarantee code integrity
- Isolation: Prevent outside attackers
- Encryption: Protect data confidentiality

TEE

- Secrets
- Confidential Computing Services
- Result
When Confidential Computing Become a Service

CCaaS Framework
CCaaS for Multiple Data Providers

Key Negotiation

Input Data Provision

Decryption

Task Execution

Encryption

Result Return

Secure Channel Establishment

Data Provider

Input Data

Not Trust

Data Provider

Output Data

CCaaS Framework

Trust but Verify

Not Trust and Verify

Task Submission

Third-party Developer
TEE’s Abilities and Inabilities

✧ **Attestation**: guarantee identity of code  
⇒ *cannot prove the trustworthiness*

✧ **Isolation**: prevent outside attackers  
⇒ *cannot prevent data leakage*

✧ **Encryption**: protect data safety  
⇒ *cannot avoid secrets withheld*
Our goal: prove to the user that the enclave service cannot threaten their private information.
**Proof of Being Forgotten (PoBF)**

- **No Leakage**: All secret and secret-tainted values are within a confined zone during computation.
- **No Residue**: After the computation (e.g., serving a user), no secret is found in the enclave.
Theoretical Foundation: Enclave Model

Table 1: Generalized model of secure enclaves.

<table>
<thead>
<tr>
<th>Type</th>
<th>Sym.</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Natural</td>
<td>$n$</td>
<td>$\in \mathbb{N}$</td>
</tr>
<tr>
<td>String</td>
<td>$str$</td>
<td>$\in S$</td>
</tr>
<tr>
<td>Bool</td>
<td>$b$</td>
<td>:= True</td>
</tr>
<tr>
<td>Value</td>
<td>$v'$</td>
<td>:= ConcreteN($n$)</td>
</tr>
<tr>
<td>Sec. Tag</td>
<td>$vt$</td>
<td>:= Secret</td>
</tr>
<tr>
<td>TagValue</td>
<td>$v$</td>
<td>:= ($v'$,$vt$)</td>
</tr>
<tr>
<td>Mode</td>
<td>$mo$</td>
<td>:= EnclaveMode</td>
</tr>
<tr>
<td>Location</td>
<td>$l$</td>
<td>:= Stack($n$)</td>
</tr>
<tr>
<td>Enc. Tag</td>
<td>$et$</td>
<td>:= Zone</td>
</tr>
<tr>
<td>Cell</td>
<td>$c$</td>
<td>:= Normal($v$)</td>
</tr>
<tr>
<td>Result</td>
<td>$r$</td>
<td>:= Ok($X$)</td>
</tr>
<tr>
<td>Error</td>
<td>$e$</td>
<td>:= Invalid</td>
</tr>
<tr>
<td>Storable</td>
<td>$me$</td>
<td>:= List($l,c$)</td>
</tr>
</tbody>
</table>

Table 2: Enclave program syntax.

<table>
<thead>
<tr>
<th>Term</th>
<th>Sym.</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exp.</td>
<td>$e$</td>
<td>:= $l$</td>
</tr>
<tr>
<td>Proc.</td>
<td>$p$</td>
<td>:= Nop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If $e$ Then $p1$ Else $p2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$p1;p2$</td>
</tr>
</tbody>
</table>
Theoretical Foundation: NoLeakage Theorem

A procedure’s execution does not leak secret.

- Its initial state is secure;
- All memory writes are within the Zone;
- It aborts when error occurs;
Theoretical Foundation: NoResidue Theorem

If the Zerorize procedure is executed at the end of a function, then no sensitive data residue is left in the enclave.

zerorize

Clears the values stored in the confined zone.
Theoretical Foundation: Checked by Coq

✓ Mechanically Checked by Coq
Realizing the secure enclave service.
Design Goals

Security:

- No Leakage
- No Residue
- Verifiable

Auxiliary:

- Minimal code modification
- Various hardware TEE support
**PoBF-Compliant Framework (PoCF)**

System Overview

Our Artifacts:
- PoCF Library (TEE-Agnostic)
- PoCF Enclave (TEE-Specific)
- PoCF Verifier

Submitted by 3rd Party Developer:
- CC (Confidential Computing) Task
Pillar of PoCF: Workflow Integrity

Control Flow Integrity
Type Safety | Memory Safety

Workflow Integrity?
Typestate Specification

✓ Specified.
✓ Enforced by Rust.
✓ Verified by Prusti.
✓ Statically checked.
✓ Finally, workflow integrity guaranteed with minor runtime cost!

Listing 1: Typestate abstraction and specification.

```rust
pub struct Task<S, K, D> where
S: TaskState + DataState + KeyState,
K: Zeroize + Default, D: EncDec<K>,
<S as DataState>::State: DState,
<S as KeyState>::State: KState,
{
  data: Data<<S as DataState>::State, D, K>,
  key: Key<K, <S as KeyState>::State>,
  _state: S,
}

pub trait TaskState {
  #[pure]
  fn is_initialized(&self) -> bool { false }
  #[pure]
  fn is_finished(&self) -> bool { false }
  // Other similar functions are omitted.
}

pub struct Initialized;
#[refine_trait_spec]
impl TaskState for Initialized {
  #[pure]
  #[ensures(result == true)]
  fn is_initialized(&self) -> bool { true }
}

#[ensures((&result)._state.is_allowed_once())]
// Other similar specifications are omitted
pub fn cc_compute(&self) ->
  Task<ResultEncrypted, Invalid, EncryptedOutput>;
```
Workflow Integrity by Rust & Typestate

Control Flow Integrity

Type Safety

Memory Safety

Workflow Integrity
NoResidue Instrumentation

✓ Heap: modified Memory Allocator
✓ Global: not mutable
✓ Stack and Registers: Instrumentation

Listing 1: Typestate abstraction and specification.

```rust
pub struct Task<S, K, D> where
S: TaskState + DataState + KeyState,
K: Zerorize + Default, D: EncDec<K>,
<S as DataState>::State: DState,
<S as KeyState>::State: KState,
{
    data: Data<<S as DataState>::State, D, K>,
    key: Key<<K, <S as KeyState>::State>,
    _state: S,
}

pub trait TaskState {  
    #[pure]
    fn is_initialized(&self) -> bool {false}
    #[pure]
    fn is_finished(&self) -> bool {false}
    // Other similar functions are omitted.
}

pub struct Initialized;
#[refine_trait_spec]
impl TaskState for Initialized {
    #[pure]
    #[ensures(result == true)]
    fn is_initialized(&self) -> bool {true}
}

#[ensures((&result)._state.is_allowed_once())]
// Other similar specifications are omitted
pub fn cc_compute(self) -> Task<ResultEncrypted,Invalid,EncryptedOutput>;
```

No Residue
NoLeakage Verification

✓ Edge function calls: does not leak secret.
  • E.g., OCALL in SGX and call to the hypervisor in SEV

✓ Static taint analysis
  • Key’s tracking: typestate
  • Data tracking: MIRAI’s taint analysis
PoCF Verifier

- Once CC Task Submitted: the deployer verifies it.

- Data providers:
  1. Obtain the source code.
  2. Conduct verification.
  3. Calculate measurement.
  4. Feed data.

- Trusted builder: proprietary code.
Evaluation
Summary of Evaluation Results

1. PoCF reaches its design goals.

2. PoCF incurs negligible overhead in CPU-bound tasks.

3. PoCF exhibits degradation in IO-bound tasks (lack of IO optimizations).

4. The data flow tracking tool is not very accurate.
Questions?
You’re welcome to try and star our artifact!

Github: ya0guang/PoBF
Thanks!
Backup Slides
PoCF Library: TEE-Agnostic State Machine

- **Initialized**
  - K: Uninitialized
  - D: Uninitialized
  - establish_channel()

- **Channel Established**
  - K: AllowedTwice
  - D: Uninitialized
  - K: AllowedTwice
  - D: EncryptedInput
  - receive_data()

- **Data Received**
  - K: AllowedOnce
  - D: EncryptedInput
  - decrypt_data()

- **Data Decrypted**
  - K: AllowedOnce
  - D: DecryptedInput
  - decrypt_data()

- **Result Encrypted**
  - K: AllowedOnce
  - D: EncryptedOutput
  - cc_compute()
  - encrypt_result()

- **Result Decrypted**
  - K: AllowedOnce
  - D: DecryptedOutput

- **Finished**
  - K: Invalid
  - D: EncryptedOutput
  - take_result()
PoCF Enclave: **TEE-Specific Enclave Service**

- Intel SGX
  - DCAP & EPIP Attestation
  - Teaclave (Rust) SGX SDK
  - ECALL & OCALL

- AMD SEV on Azure
  - Azure Attestation Service
  - Standard Library
Effortless Porting

- Verifier invocations wrapped.
- Seamless use of standard library
## Taint Analysis: Accuracy of MIRAI

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

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

All the tests were analyzed by MIRAI using its strictest analysis level, i.e., MIRAI_FLAG=diag=paranoid.

✓: No data leakage; X: Has data leakage; o: Possible data leakage. The number behind “X” or “o” denotes the number of data leakages.


Microbenchmark: Polybench

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

(b) Polybench: Performance of PoCF and NATIVE on SEV.
Microbenchmark: Overhead Analysis

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

<table>
<thead>
<tr>
<th>Config</th>
<th>1KB</th>
<th>10KB</th>
<th>100KB</th>
<th>1MB</th>
<th>10MB</th>
<th>100MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>NATIVE X</td>
<td>275.8</td>
<td>281.1</td>
<td>296.3</td>
<td>536.7</td>
<td>3026.5</td>
<td>28018.3</td>
</tr>
<tr>
<td>P w/o T X</td>
<td>278.3</td>
<td>280.4</td>
<td>298.6</td>
<td>541.1</td>
<td>3033.9</td>
<td>28022.9</td>
</tr>
<tr>
<td>P w/ T X</td>
<td>277.3</td>
<td>287.4</td>
<td>301.7</td>
<td>545.0</td>
<td>3043.7</td>
<td>28215.0</td>
</tr>
<tr>
<td>NATIVE V</td>
<td>489.1</td>
<td>487.3</td>
<td>449.7</td>
<td>495.6</td>
<td>502.0</td>
<td>923.3</td>
</tr>
<tr>
<td>PoCF V</td>
<td>489.5</td>
<td>492.3</td>
<td>454.4</td>
<td>499.8</td>
<td>506.5</td>
<td>934.8</td>
</tr>
</tbody>
</table>

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

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

Figure 5: Identity task: Performance breakup of PoCF.
Macrobenchmark: AI Inference

![Graph showing comparison between different systems in terms of execution time for AI inference tasks.](image)

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

Figure 7: Macrobenchmark: AI inference execution time.
Macrobenchmark: FASTA

Figure 8: Macrobenchmark: FASTA execution time.
Macrobenchmark: In-memory KVDB

(a) Single-Thread Latency.
(b) Single-Thread Throughput.
(c) Multi-Thread Latency.
(d) Multi-Thread Throughput.