Proof of Being Forgotten

Verified Privacy Protection in Confidential Computing Platform

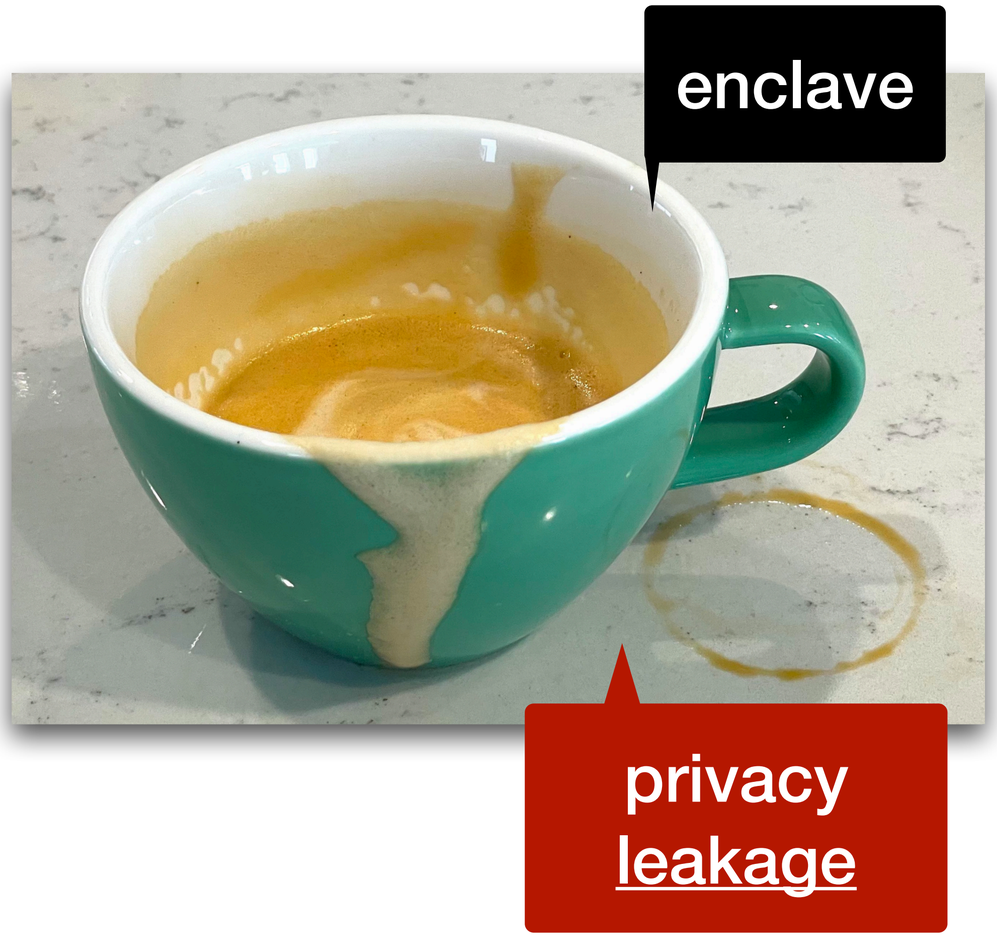
Mingshen Sun, Hongbo Chen, Zhaofeng Chen, Kang Li

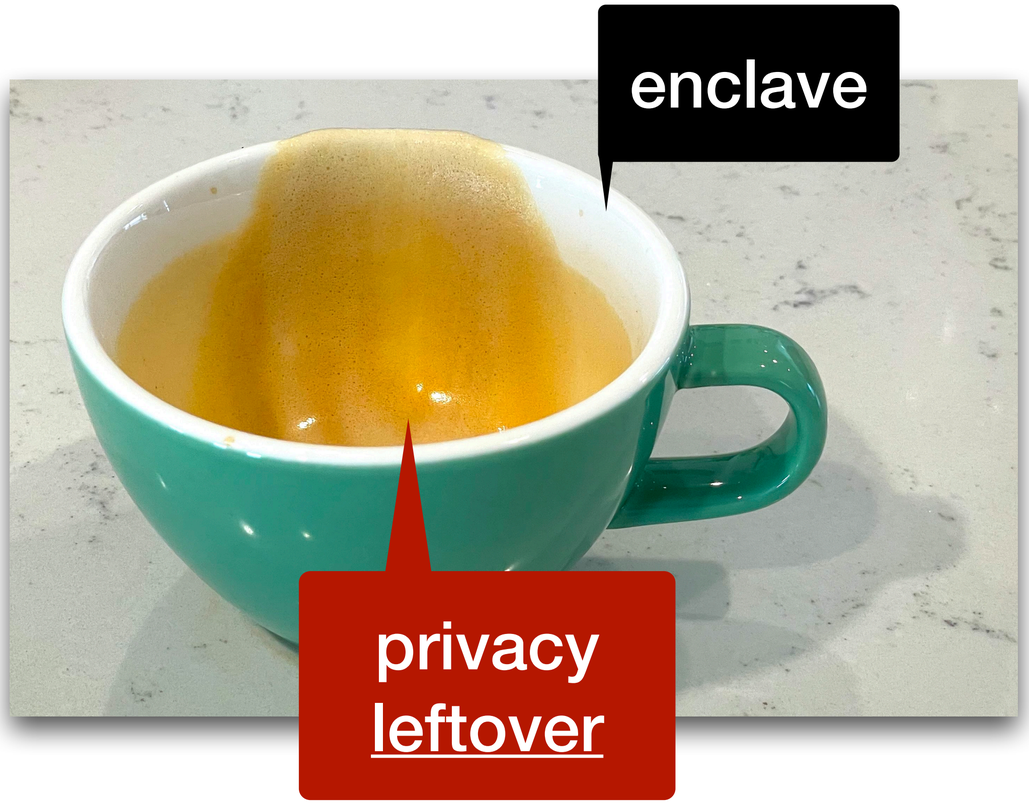
Baidu

Open Confidential Computing Conference 2022

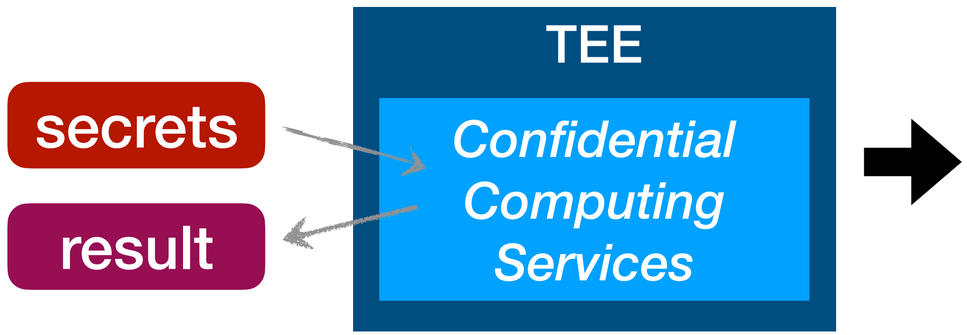
Coffee Incidents

Privacy Incidents





TEE's Abilities and Inabilities



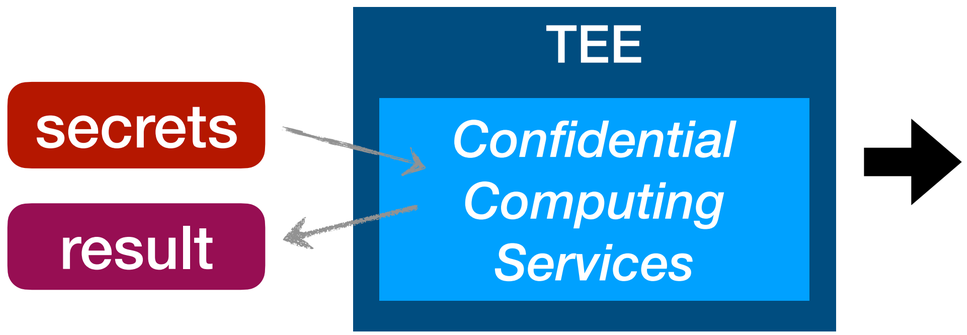
✓ **Attestation**: guarantee identity of code

✓ **Isolation**: prevent outside attackers

✓ **Encryption**: protect data safety

TEE's Abilities and Inabilities

TEE doesn't not guarantee no data **leakage** and secrets **withheld**



✓ **Attestation**: guarantee identity of code

✦ **Isolation**: ~~prevent outside attackers~~

⇨ *cannot prevent data leakage*

✦ **Encryption**: ~~protect data safety~~

⇨ *cannot avoid secrets withheld*

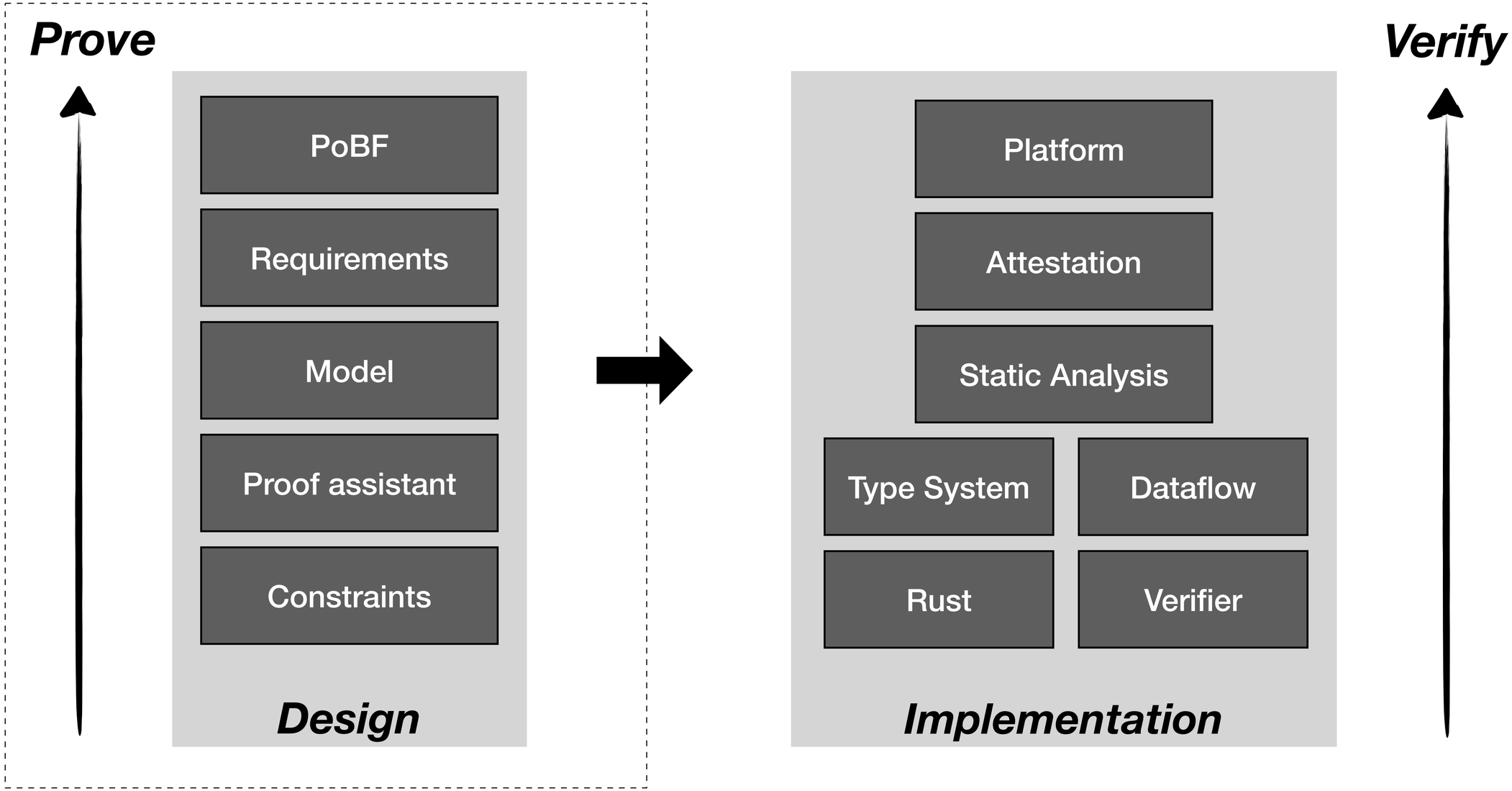
Proof of Being Forgotten

a principal regulates the code dealing with secrets is verified so that secrets are completely consumed and not revealed to any unauthorized party

✓ **NO LEAKAGE**

✓ **NO RESIDUE**

Towards Proof of Being Forgotten

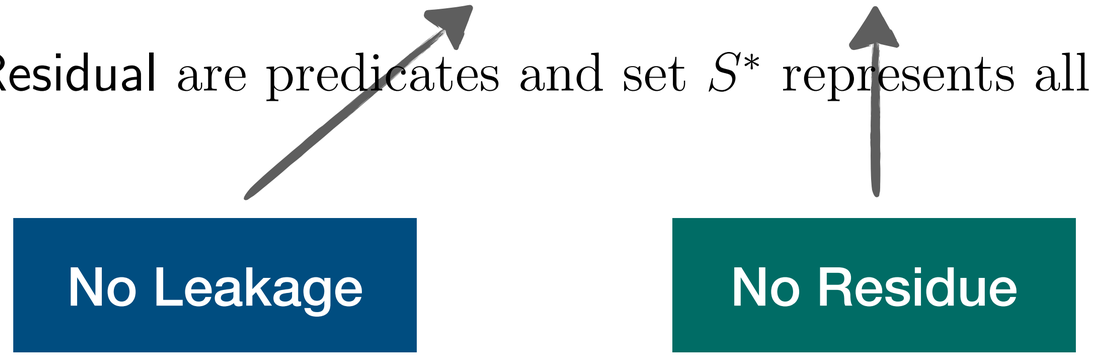


PoBF Formal Definition

Definition 1.2 (Being Forgotten for Procedure)**.** For a critical procedure $p(S)$, the Being Forgotten predicate is defined as follows:

$$\mathsf{BF}(p(S)) := \neg\mathsf{Leaked}(p, S^*) \wedge \neg\mathsf{Residual}(p, S^*)$$

where Leaked and Residual are predicates and set S^* represents all values tainted by $s \in S$.



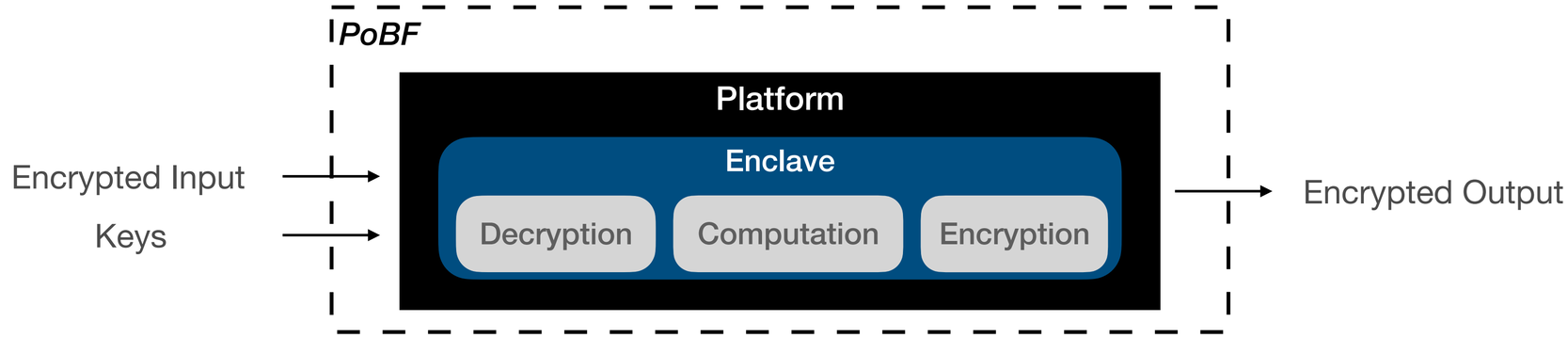
PoBF-Compliant Requirements

$$\mathrm{BF}(T) \Leftrightarrow \bigwedge_{p_i \in T} \mathrm{Restricted}(p_i(S), \mathcal{Z}) \wedge \bigwedge_{p_i \in T} \mathrm{CleanMoved}(p_i(S), \mathcal{Z}) \wedge \mathrm{EndWithEncryption}(T)$$

No Residue

No Leakage

Compliance



Formal Constraints

We restrict write to the zone when the procedure is running, and denote this rule as $\mathsf{Restricted}(p(S), \mathcal{Z})$. More specifically, the procedure can only write to this zone($\mathsf{WriteBounded}$) and such zone cannot be read by other procedures($\mathsf{ReadExclusive}$):

$$\mathsf{Restricted}(p(S), \mathcal{Z}) = \mathsf{WriteBounded}(p(S), \mathcal{Z}) \wedge \mathsf{ReadExclusive}(p(S), \mathcal{Z})$$

$$\mathsf{WriteBounded}(p(S), \mathcal{Z}) \Leftrightarrow \forall op(arg*) \in p.\ op(arg*) = write(arg*) \Rightarrow args \in \mathcal{Z}$$

$$\mathsf{ReadExclusive}(p(S), \mathcal{Z}) \Leftrightarrow \forall op(arg*) \in T \setminus \{p\}.\ op(arg*) = read(arg*) \Rightarrow args \notin \mathcal{Z}$$

`WriteBounded`:
Dataflow Analysis in MIRAI

`SafeTransferred`:
Typestate in Rust

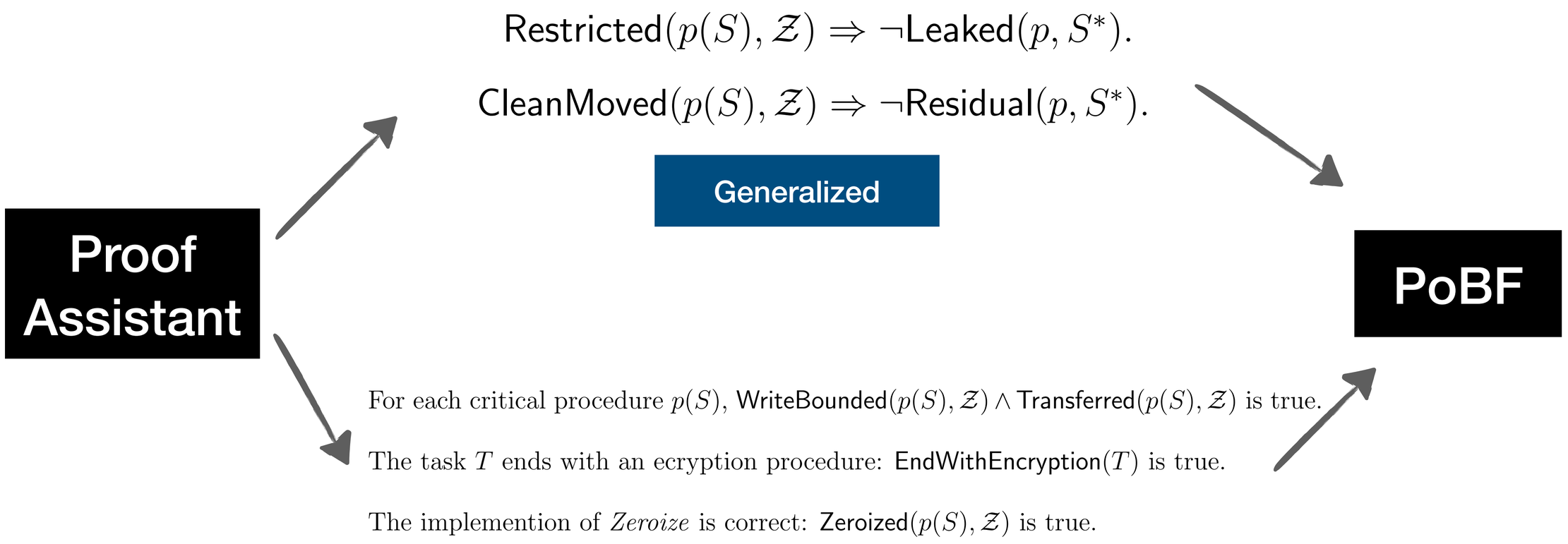
For the residue, we enforce a safe cleaning at the end of p and denote as rule $\mathsf{CleanMoved}(p(S), \mathcal{Z})$. Zeros are written to all memory locations in $\mathcal{Z}$ other than the return value r_p after the execution of $p(S)$ ($\mathsf{Zeroized}$ rule). Besides, $\mathsf{SafeTransferred}$ rule stipulates that the return value r_p must be passed to the next procedure intactly, and cannot be accessed by any other entities. Formally speaking,

$$\mathsf{CleanMoved}(p(S), \mathcal{Z}) = \mathsf{Zeroized}(p(S), \mathcal{Z}) \wedge \mathsf{SafeTransferred}(p(S), p_{next}(S_{next}))$$

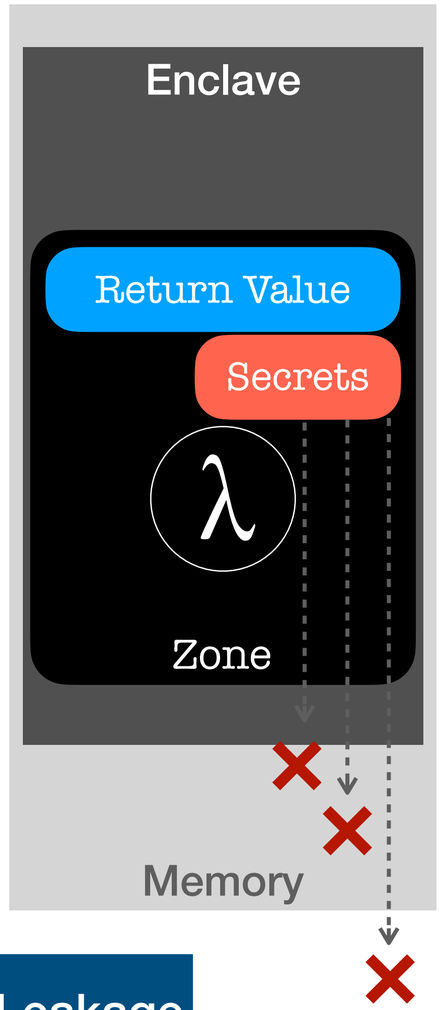
$$\mathsf{Zeroized}(p(S), \mathcal{Z}) \Leftrightarrow \forall l \in \mathcal{Z} \setminus loc(r_p).\ read(l) = 0$$

$$\mathsf{SafeTransferred}(p(S), p_{next}(S_{next})) \Leftrightarrow r_p \in S_{next} \wedge (\forall op \in T, arg \in loc(r_p).\ op = read(arg*) \Rightarrow op \in p_{next}(S_{next}))$$

PoBF Formal Foundation



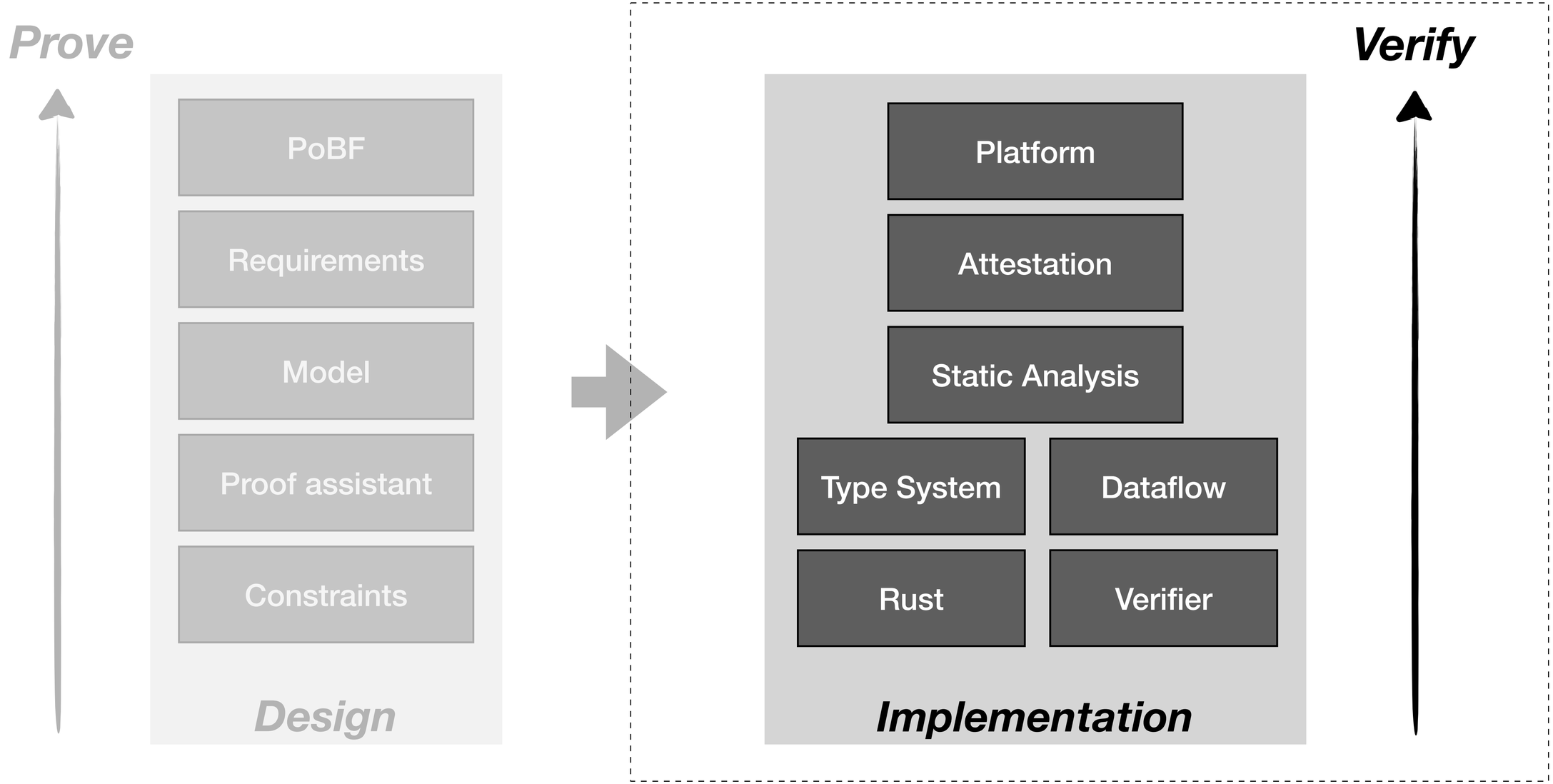
Memory Model



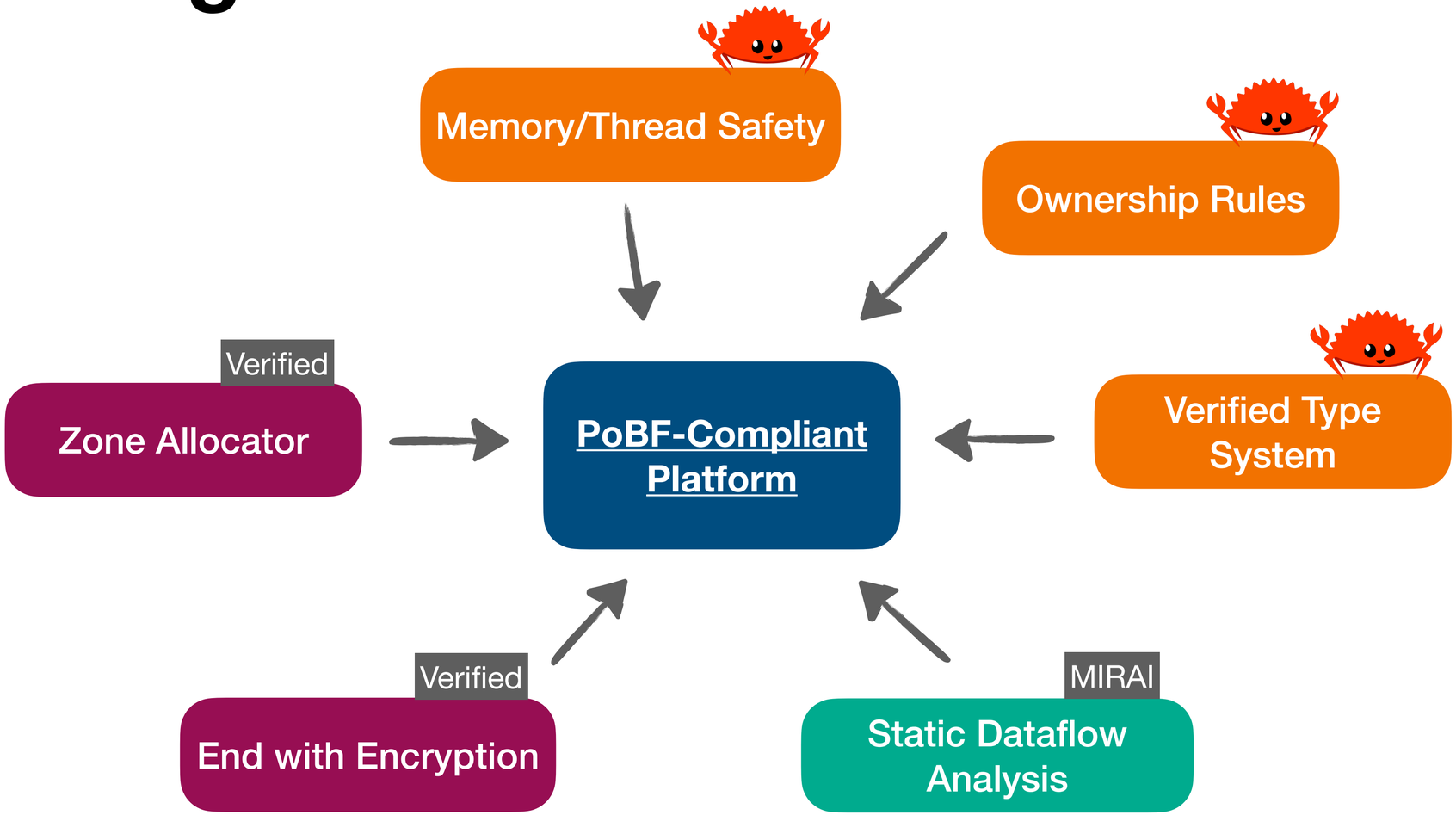
No Leakage

No Residue

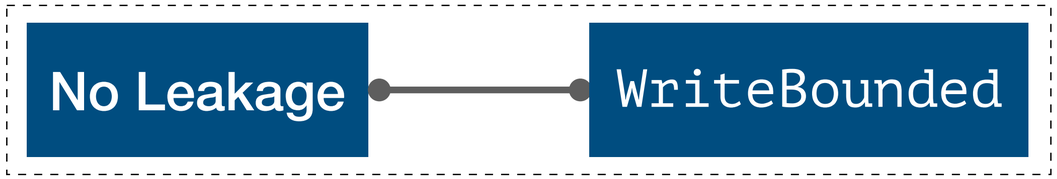
Towards Proof of Being Forgotten



A Verified Privacy Protection in Confidential Computing Platform



Example: Detect Secret Leakage by Static Dataflow Analysis



Tag

Log

Verify

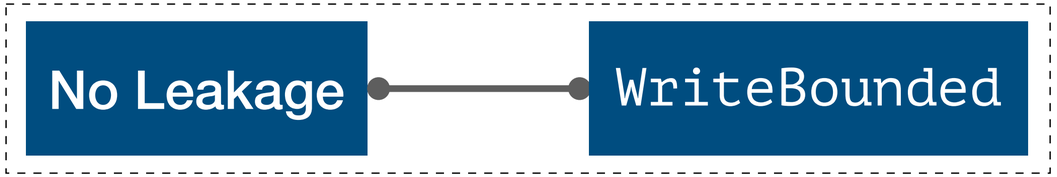
```
31      let protected = ProtectedAssets::new(
32          secret_data,
33          input_key,
34          output_key,
35      );
36      add_tag!(&protected, SecretTaint);
37
38      // warn by mirai
39      safe_log(&protected.output_key.raw);
40      // not warn
41      safe_log(&"computation start.".as_bytes().to_vec());
```

```
92 fn safe_log(input: &Vec::<u8>) {
93      precondition!(does_not_have_tag!(input, SecretTaint));
94      println!("{}", String::from_utf8_lossy(input));
95 }
```

Leakage Detected by MIRAI

```
→  PoBF git:(talk) cargo mirai
warning: unsatisfied precondition
  --> src/main.rs:39:5
   |
39 |      safe_log(&protected.output_key.raw);
   |      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
note: related location
  --> src/main.rs:93:5
   |
93 |      precondition!(does_not_have_tag!(input, SecretTaint));
   |      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   = note: this warning originates in the macro `precondition`
(in Nightly builds, run with -Z macro-backtrace for more info)
warning: `pobf` (bin "pobf") generated 1 warning
    Finished dev [unoptimized + debuginfo] target(s) in 0.54s
```

Example: Detect Secret Leakage by Static Program Analysis




```
$ cat enclave/src/userfunc.rs
#![forbid(unsafe_code)]

use crate::pobf_verifier::*;

pub fn vec_inc(input: Vec<u8>) -> Vec<u8> {
    let step = 1;

    // This is accepted by PoBF Verifier.
    println!("The step is {} in computation_enc", step);

    let mut output = Vec::new();
    for i in input.iter() {
        output.push(i + step);
    }

    // However, PoBF Verifier complians about this print log.
    // Violation: cannot log the secret data
    println!("after increasing, the 0th data is {}", output[0]);

    output
}
```

Log secret data!

Leakage Detected by PoBF Verifier

```
$ cargo pobf-verify
INFO checking if the source code forbids `unsafe` ...
INFO checking unsafe code...
INFO - analyzing file: ./src/types/vecaes.rs...
INFO   + unsafe code fobidden in ./src/types/vecaes.rs, clear.
...
INFO checking possible leakage through OCALLs...
ERROR PoBF Verifier found possible leakage.
WARNING The detailed explaination of this error type [vio-ocall] can be found
in the document.
WARNING: possible false verification condition
--> src/userfunc.rs:19:9
   |
   |         verify!(output[0] == 43);
   |         ^^^^^^^^^^^^^^^^^^^^^^^^
19 |         println!("after increasing, the 0th data is {}", output[0]);

ERROR `no_leakage` verification failed: 1 leakage(s) found.
```

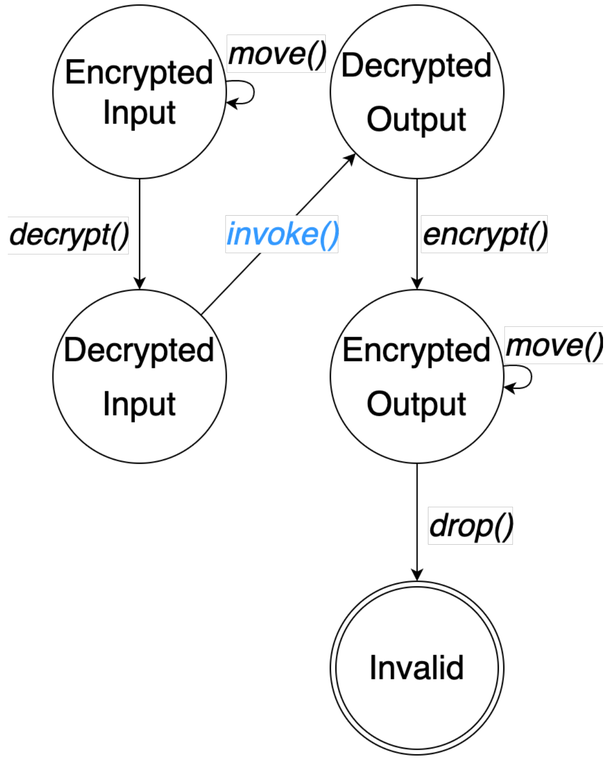
Example: Restrict State Transition by Typestate in Rust

ProtectedAssets<Decrypted, **Input**> —invoke()→ ProtectedAssets<Decrypted, **Output**>

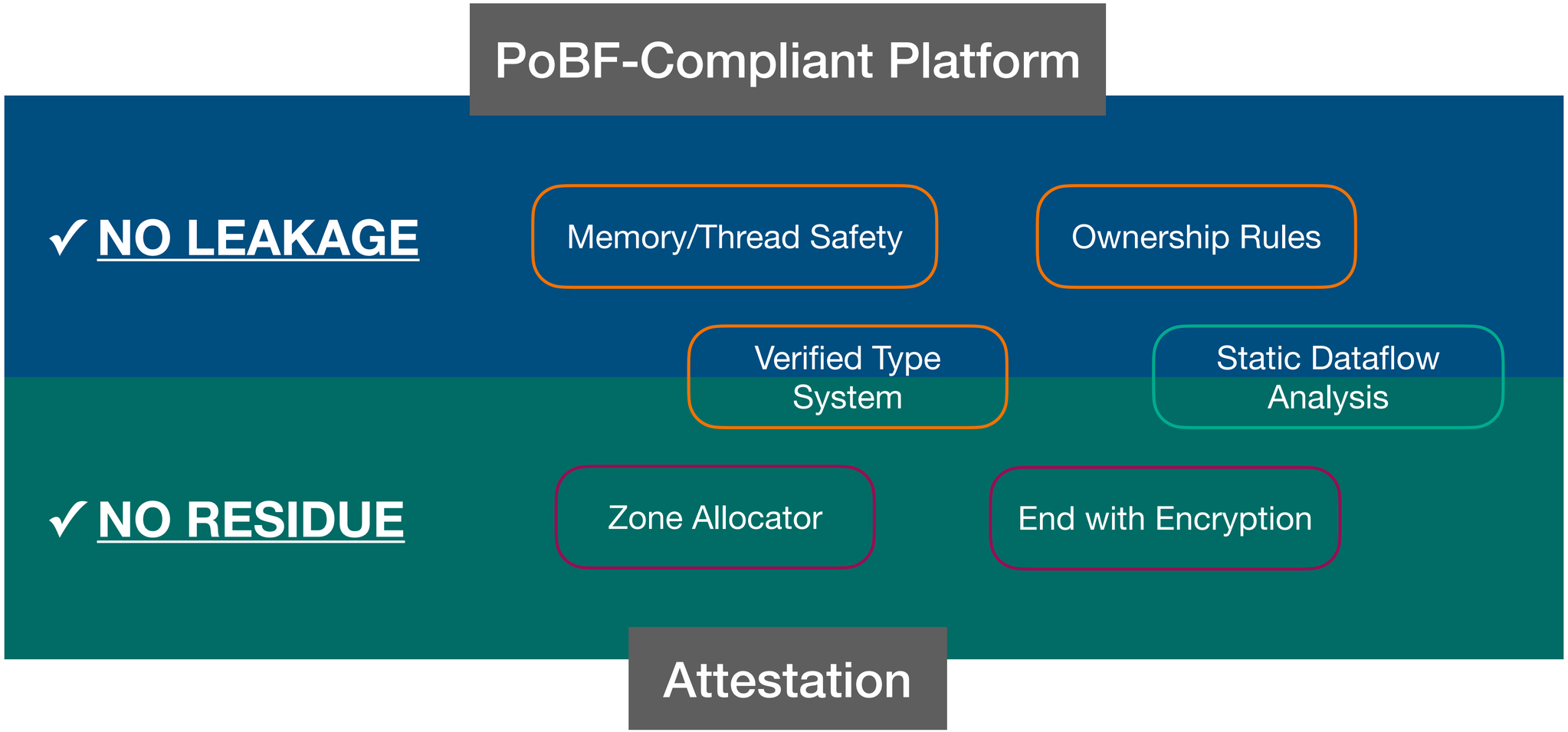
ProtectedAssets<**Decrypted**, Output> —encrypt()→ ProtectedAssets<**Encrypted**, Output>

```rust
impl ProtectedAssets<Decrypted, Input> {
    pub fn invoke(self, fun: &dyn Fn(Vec<u8>) -> Vec<u8>) ->
ProtectedAssets<Decrypted, Output> {
        ProtectedAssets {
            data: self.data.invoke(fun),
            input_key: self.input_key,
            output_key: self.output_key,
        }
    }
}

impl ProtectedAssets<Decrypted, Output> {
    pub fn encrypt(self) -> ProtectedAssets<Encrypted, Output> {
        ProtectedAssets {
            data: self.data.encrypt(&self.output_key),
            input_key: self.input_key,
            output_key: self.output_key.zeroize(),
        }
    }
}
```



Proof of Being Forgotten

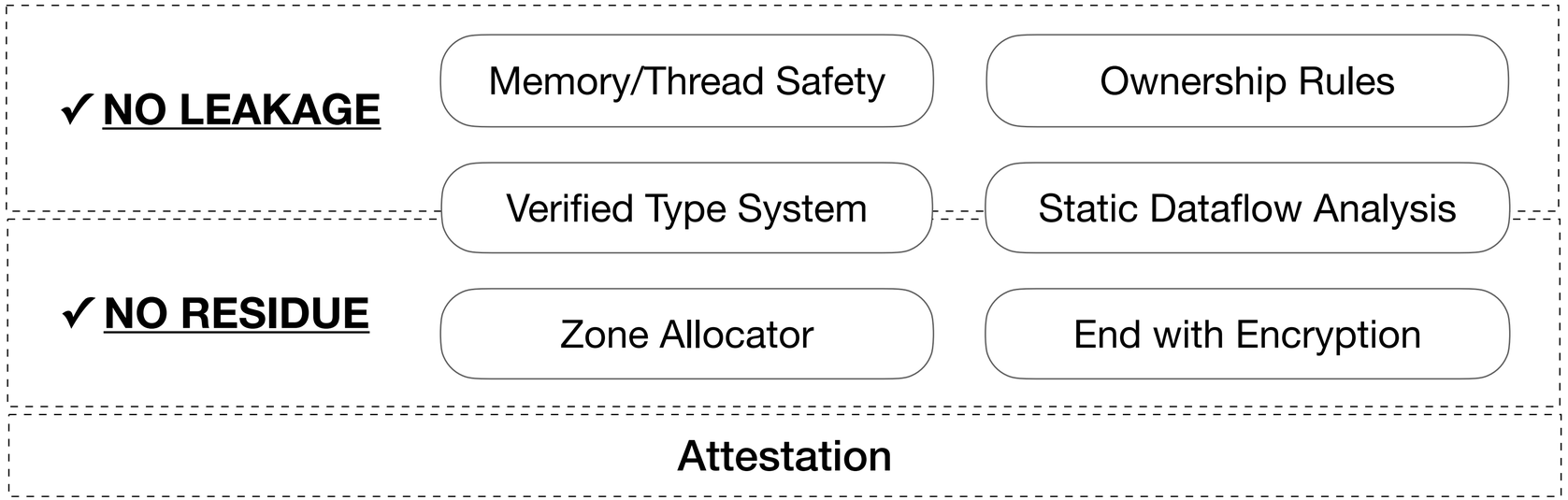


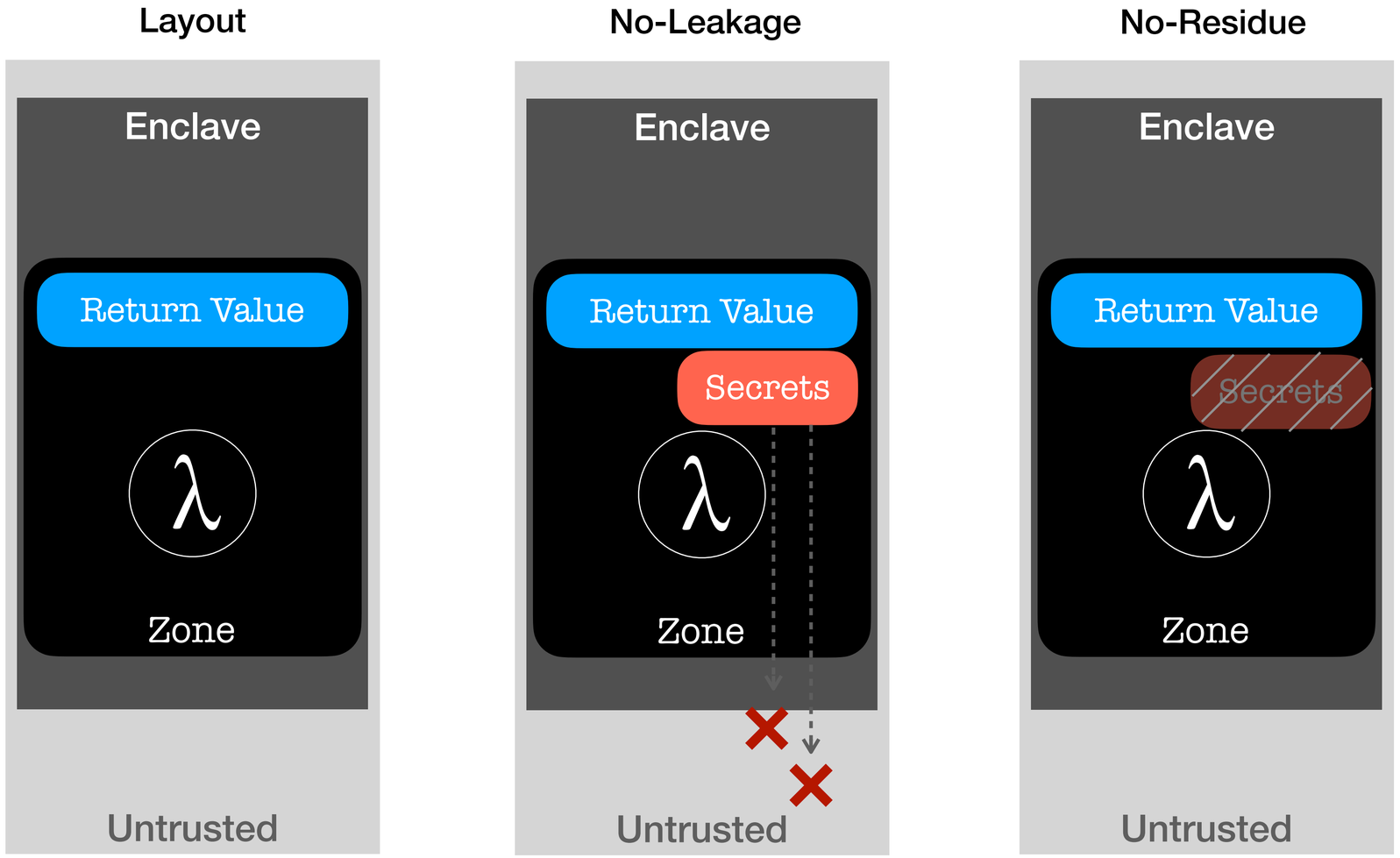
Summary

- **Proof of Being Forgotten**: a principal regulates the code dealing with secrets is verified so that secrets are completely consumed and not revealed to any unauthorized party.
- **Two requirements** of PoBF: No Leakage & No Residue
- **Formal constraints** to prove PoBF requirements.
- **A concrete design** of PoBF-compliant platform for general confidential computing.
- **Verified privacy protection platform implementation**: statically verified by the Rust type system and dataflow analysis. With the remote attestation, end users can trust the code of the platform.
- **P4Cleanroom**: our privacy-preserving computational biology platform with the PoBF property.

Thanks!

Backup


✓ NO LEAKAGE
Memory/Thread Safety
Ownership Rules
Verified Type System
Static Dataflow Analysis
✓ NO RESIDUE
Zone Allocator
End with Encryption
Attestation


Layout
Enclave
Return Value
λ
Zone
Untrusted
No-Leakage
Enclave
Return Value
Secrets
λ
Zone
Untrusted
No-Residue
Enclave
Return Value
Secrets
λ
Zone
Untrusted

FAQ

Side-channel and covert-channel?

We restrict write to the zone when the procedure is running, and denote this rule as $\mathsf{Restricted}(p(S), \mathcal{Z})$. More specifically, the procedure can only write to this zone($\mathsf{WriteBounded}$) and such zone cannot be read by other procedures($\mathsf{ReadExclusive}$):

$$\mathsf{Restricted}(p(S), \mathcal{Z}) = \mathsf{WriteBounded}(p(S), \mathcal{Z}) \wedge \mathsf{ReadExclusive}(p(S), \mathcal{Z})$$

$$\mathsf{WriteBounded}(p(S), \mathcal{Z}) \Leftrightarrow \forall op(arg*) \in p.\ op(arg*) = write(arg*) \Rightarrow args \in \mathcal{Z}$$

$$\mathsf{ReadExclusive}(p(S), \mathcal{Z}) \Leftrightarrow \forall op(arg*) \in T \setminus \{p\}.\ op(arg*) = read(arg*) \Rightarrow args \notin \mathcal{Z}$$

covert-channel

∧ SecretIndependent(p(S))

∧ NonInterference(p(S))

side-channel

Example: Typestate in Rust


```rust
pub trait InputKeyState {
    type KeyState;
}

impl InputKeyState for Data<Encrypted, Input> {
    type KeyState = Sealed;
}

impl InputKeyState for Data<Decrypted, Input> {
    type KeyState = Invalid;
}

impl InputKeyState for Data<Decrypted, Output> {
    type KeyState = Invalid;
}

impl InputKeyState for Data<Encrypted, Output> {
    type KeyState = Invalid;
}
```

```rust
pub struct ProtectedAssets<S, T>
where
    Data<S, T>: InputKeyState,
    Data<S, T>: OutputKeyState,
    S: EncryptionState,
    T: IOState,
{
    pub data: Data<S, T>,
    pub input_key: Key<<Data<S, T> as InputKeyState>::KeyState>,
    pub output_key: Key<<Data<S, T> as OutputKeyState>::KeyState>,
}
```

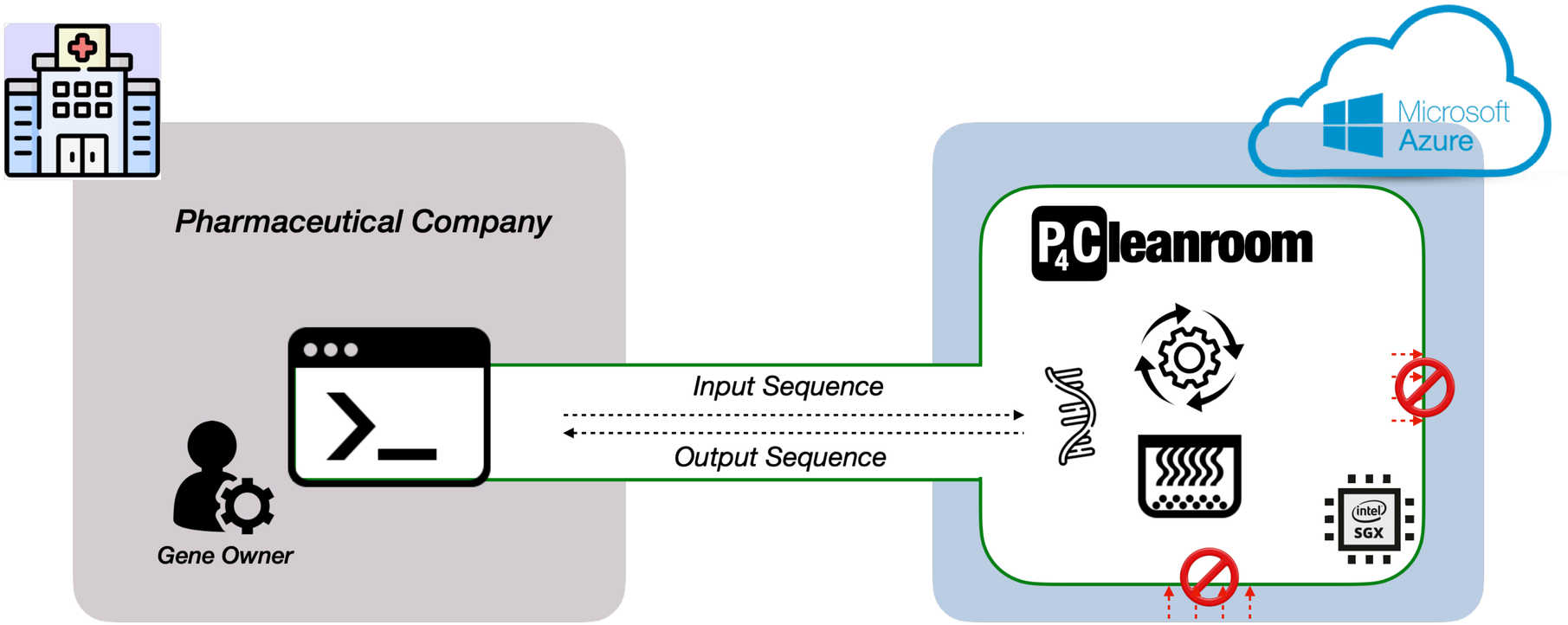
Threat Model & Assumptions

- TEE: not vulnerable
- enclave: single-threaded execution
- Rust type system and verifier: sound
- encryption/decryption: no side effect
- side/covert-channel: out of scope

Why
For Data Providers

- Data containing secrets should be handled properly in the enclave
- Service providers usually claims that user data will not be stored or used in other places.
- However, they have no way to **confirm** what is said by the service provider.
- They would like a proof that their data is really deleted(consumed) and not leaked.

P4Cleanroom: A Verified Platform Towards PoBF



P4Cleanroom is a confidential cloud service for hosting computational biology algorithms as SaaS services on the cloud. Computational biology researchers can publish their algorithms on P4Cleanroom as a SaaS service to customers, such as researchers in pharmaceutical companies and health care institutes.

https://p4cleanroom.com