Aalto University

UNIVERSITY OF WATERLOO

BliMe:
Verifiably Secure Outsource Computation with Hardware-Enforced Taint Tracking

Hossam ElAtali, *Lachlan J. Gunn, Hans Liljestrand, N. Asokan*

elatalhm.github.io

Scenario: outsourced computation

Goal: run the server's confidential code over client's confidential data

- Initial target: Outsourced ML inference and/or training

Scenario: outsourced computation

Goal: run the server's confidential code over client's confidential data

- Initial target: Outsourced ML inference and/or training

Scenario: outsourced computation

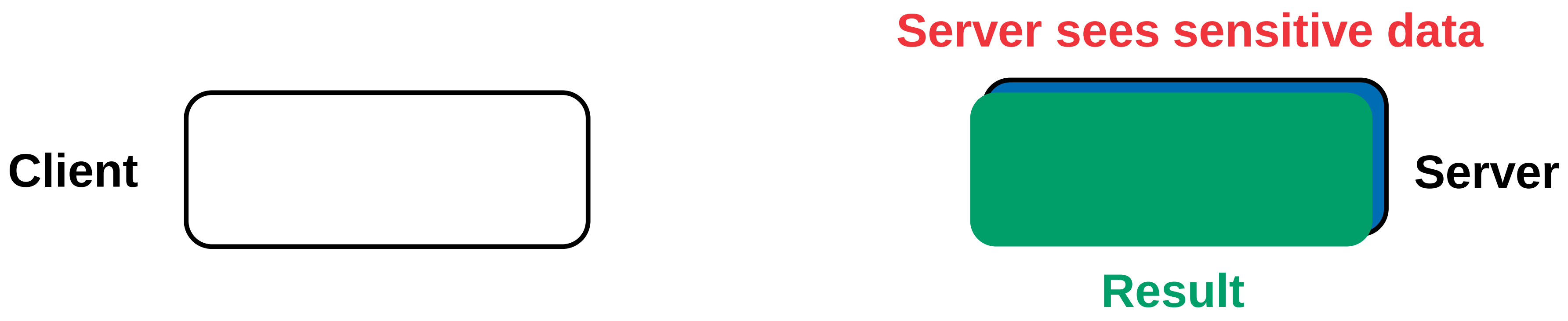
Goal: run the server's confidential code over client's confidential data

- Initial target: Outsourced ML inference and/or training

Scenario: outsourced computation

Goal: run the server's confidential code over client's confidential data

- Initial target: Outsourced ML inference and/or training



Scenario: outsourced computation

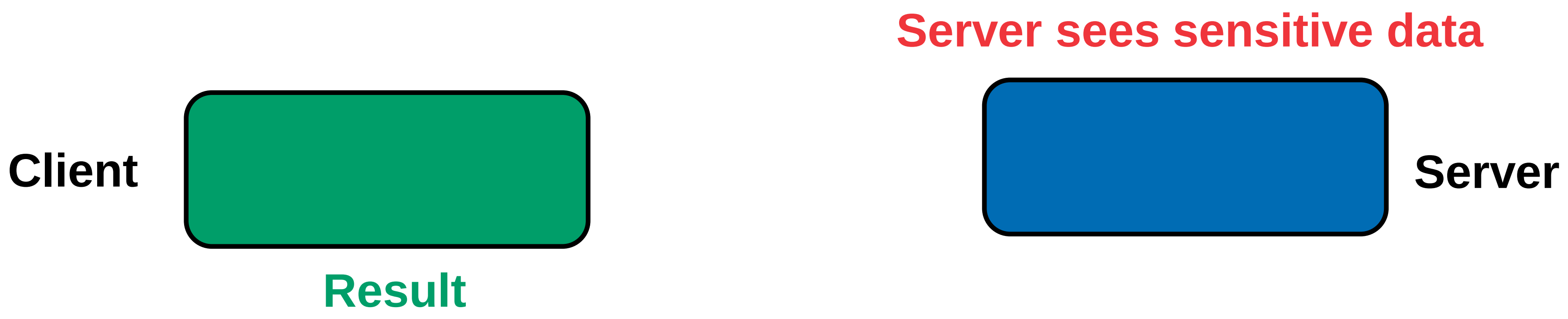
Goal: run the server's confidential code over client's confidential data

- Initial target: Outsourced ML inference and/or training

Scenario: outsourced computation

Goal: run the server's confidential code over client's confidential data

- Initial target: Outsourced ML inference and/or training



How can the client avoid revealing data to the service provider?

- Fully-Homomorphic Encryption: slow due to computational overhead
- Multi-Party Computation: slow due to network overhead

Scenario: outsourced computation

Goal: run the server's confidential code over client's confidential data

- Initial target: Outsourced ML inference and/or training

How can the client avoid revealing data to the service provider?

- Fully-Homomorphic Encryption: slow due to computational overhead
- Multi-Party Computation: slow due to network overhead

- **Hardware-based isolation + remote attestation: fast**

Protection provided by TEEs comes with caveats

TEEs provide an isolated environment for execution of software

Protection provided by TEEs comes with caveats

TEEs provide an isolated environment for execution of software

TEEs are unsuitable when server code is confidential or unverifiable

- TEEs intended for clients to run code they trust and can verify

Protection provided by TEEs comes with caveats

TEEs provide an isolated environment for execution of software

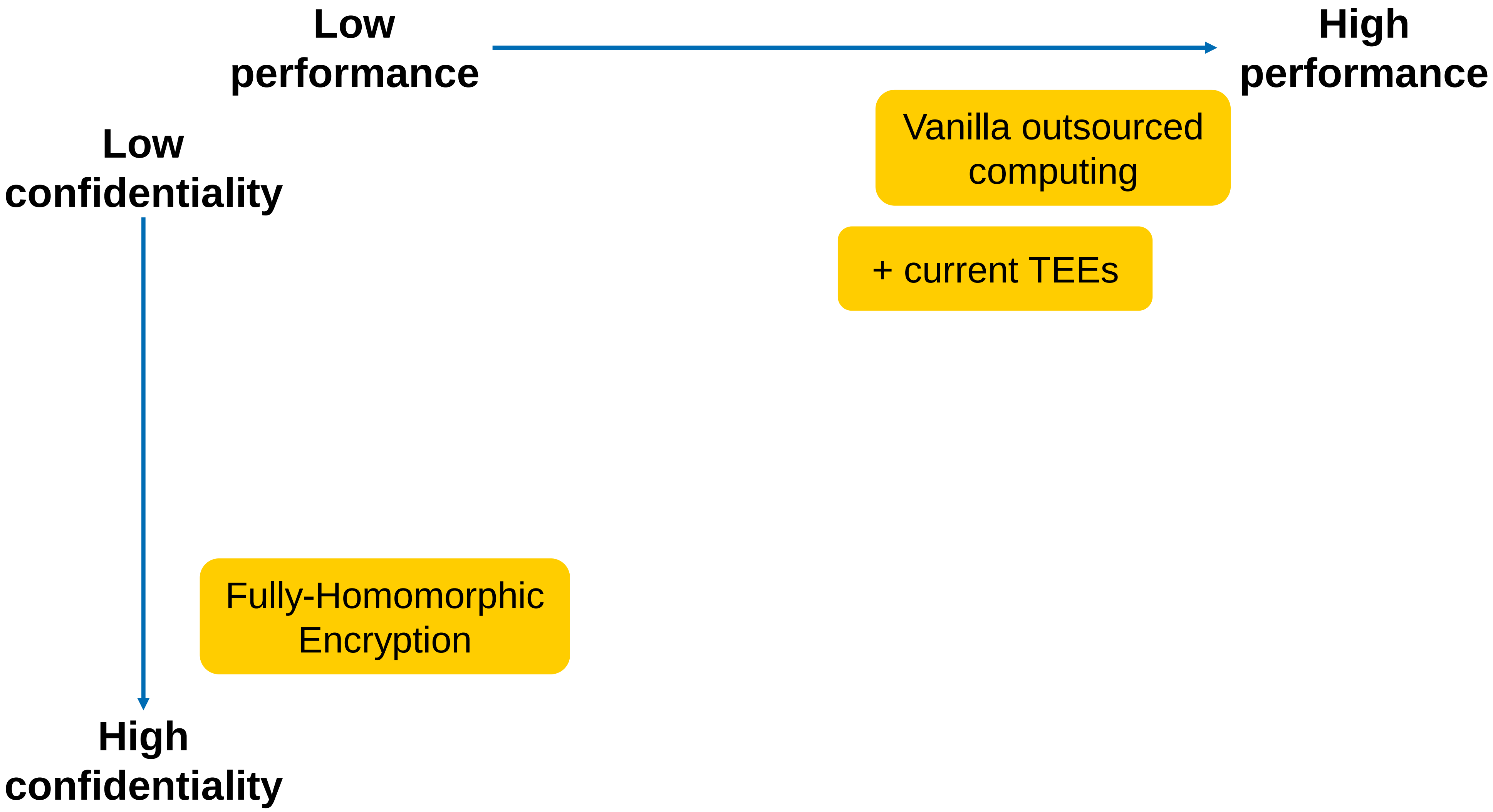
TEEs are unsuitable when server code is confidential or unverifiable

- TEEs intended for clients to run code they trust and can verify

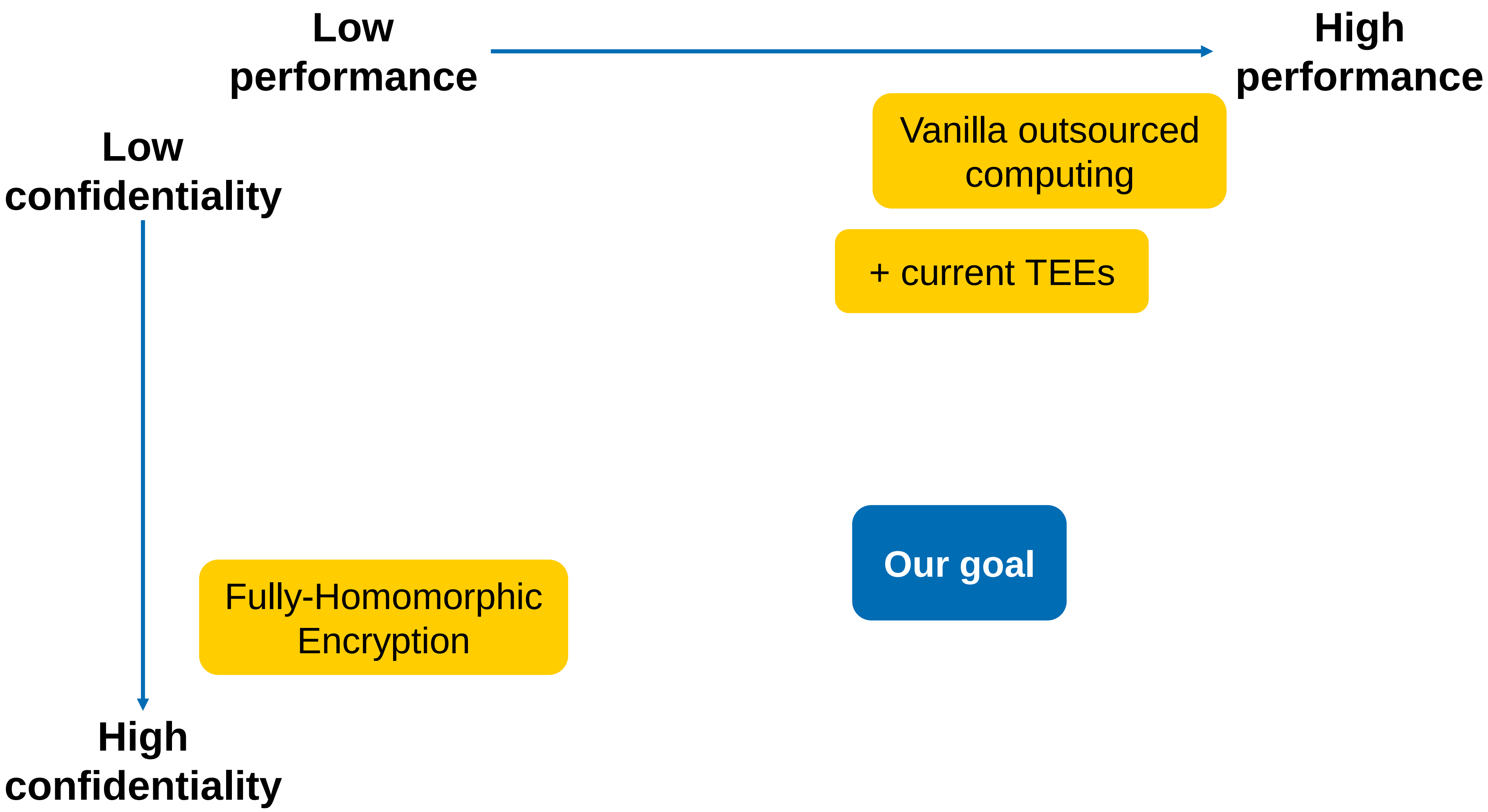
Confidentiality of client data in TEEs is hampered by:

- **Large TEE code base → vulnerable to software flaws**
- **Sharing resources → vulnerable to side channels**

Is Confidentiality vs. Performance a tradeoff?



Is Confidentiality vs. Performance a tradeoff?



What can be done?

1. **Prevent application software from leaking sensitive data**
   - Use hardware-assisted taint-tracking
   - Need not verify trustworthiness of application s/w

2. **Minimize resource sharing**
   - Move critical operations to a fixed-function, isolated module (HSM)
   - All HSM code analyzed in advance, guaranteed not to be malicious

Prevent leakage of sensitive data via CPU extensions

“Safe” streams of instructions don’t expose sensitive data

Allowed:

- Computation on sensitive data by arbitrary, unattested, untrusted software

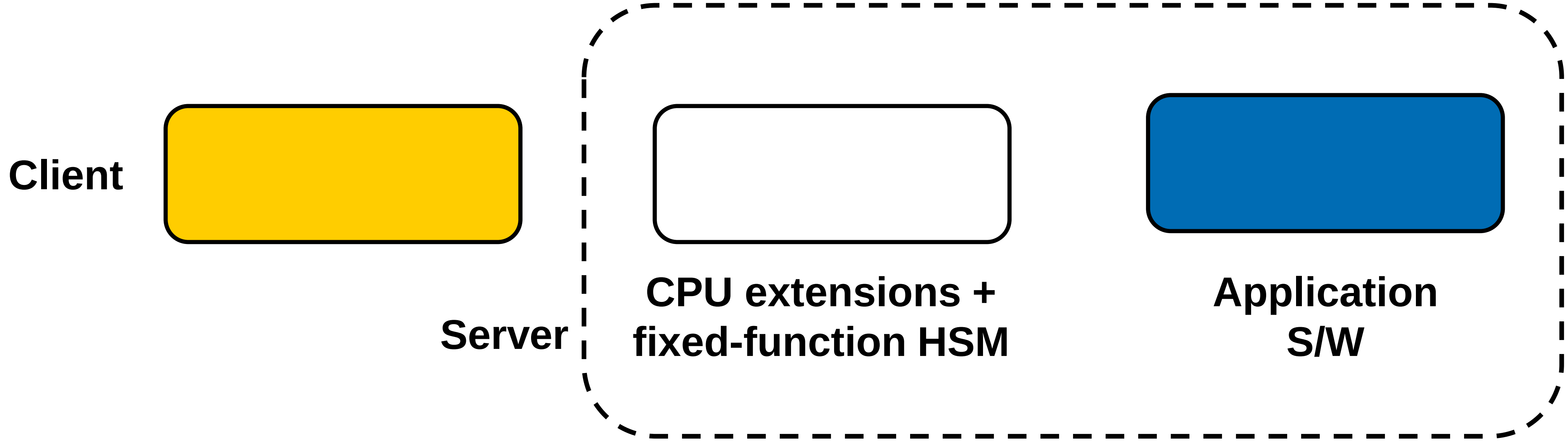
Prohibited:

- Leaking sensitive data into any observable state, e.g.: peripherals outside security boundary, microarchitectural state

Use taint-tracking-based security policy to limit sensitive data to safe places

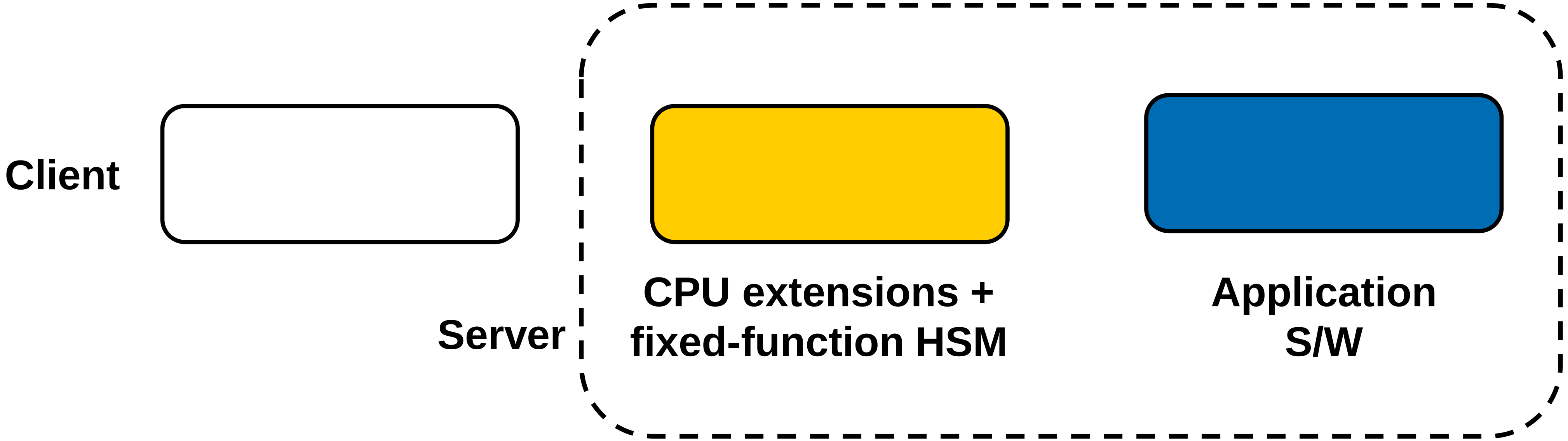
Combine with attestable HSM to assure clients

Remote attestation assures use of client data is **subject to security policy**



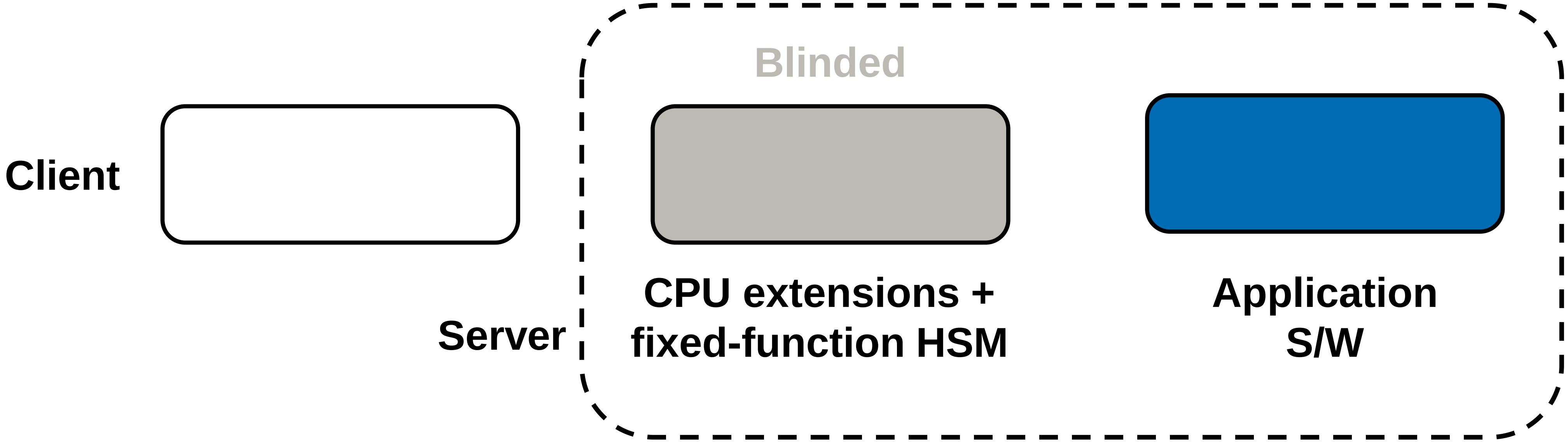
Combine with attestable HSM to assure clients

Remote attestation assures use of client data is subject to security policy



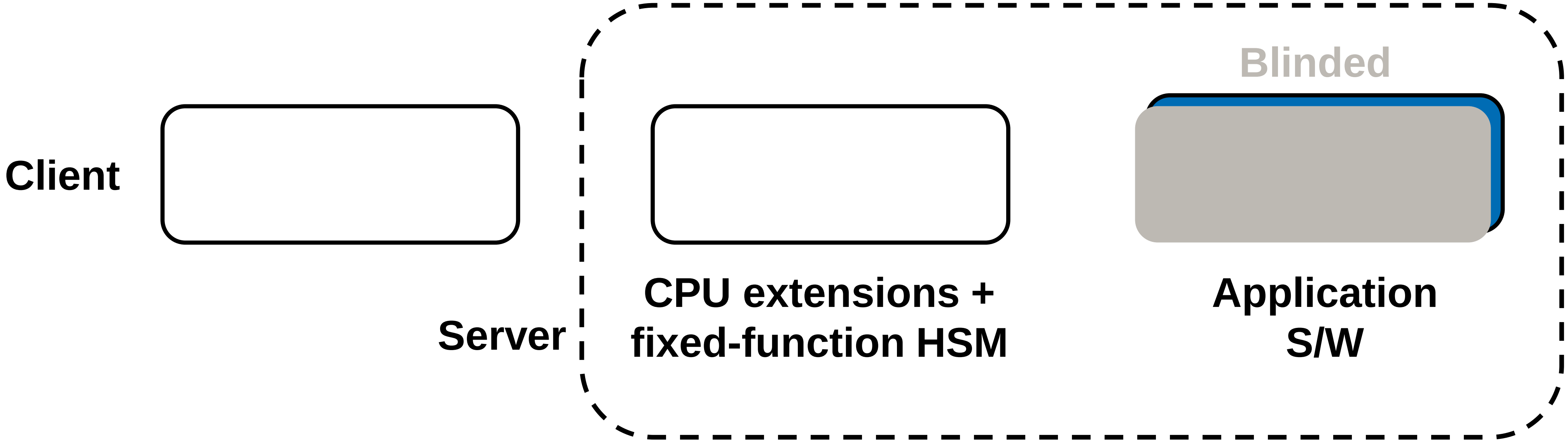
Combine with attestable HSM to assure clients

Remote attestation assures use of client data is subject to security policy



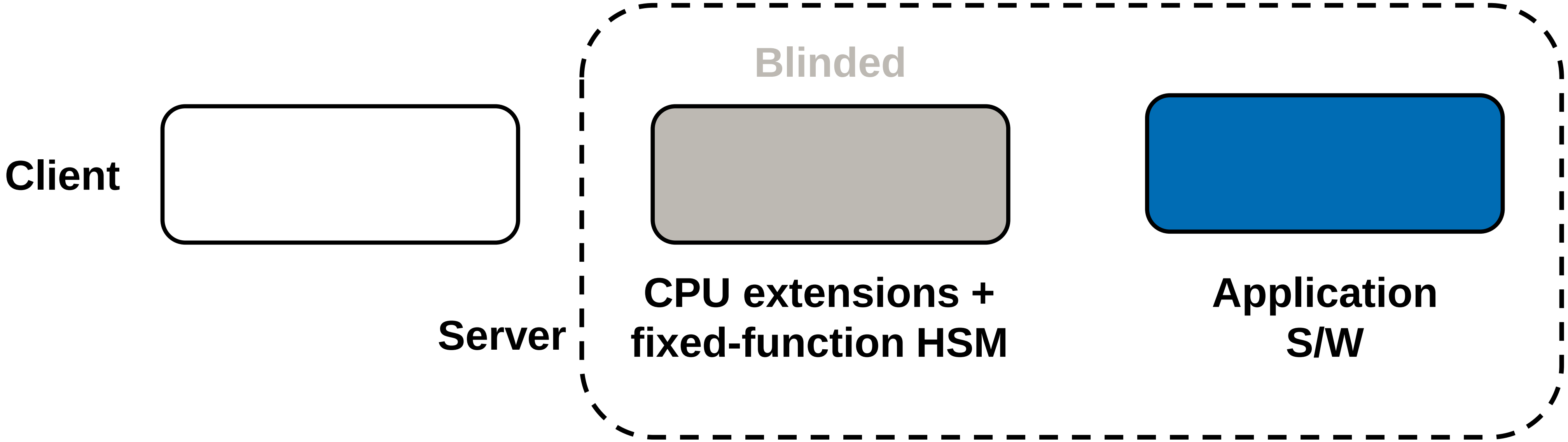
Combine with attestable HSM to assure clients

Remote attestation assures use of client data is subject to security policy



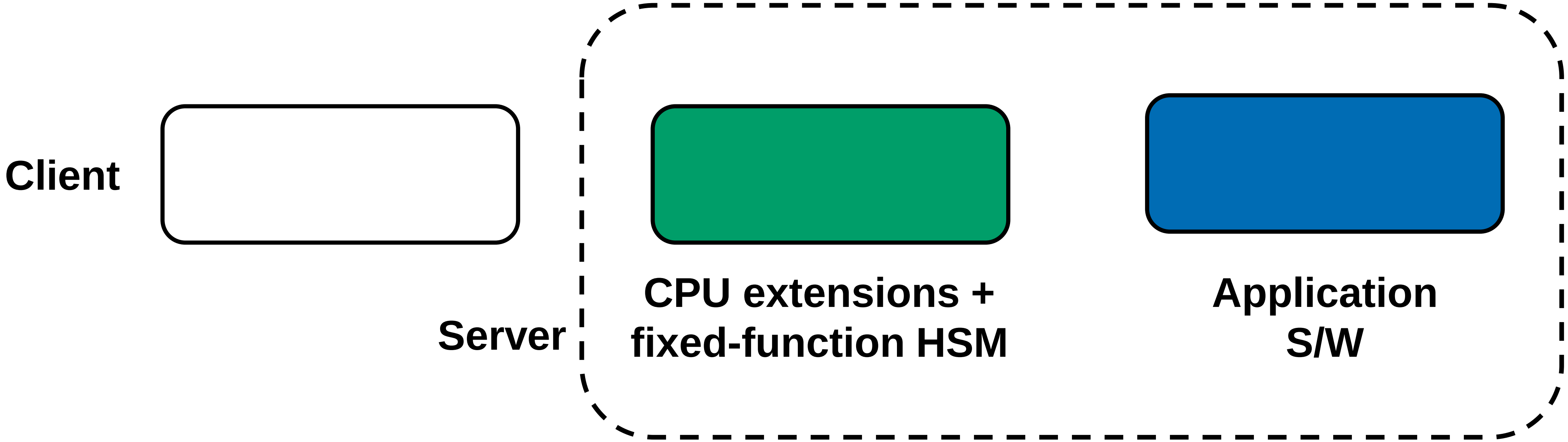
Combine with attestable HSM to assure clients

Remote attestation assures use of client data is subject to security policy



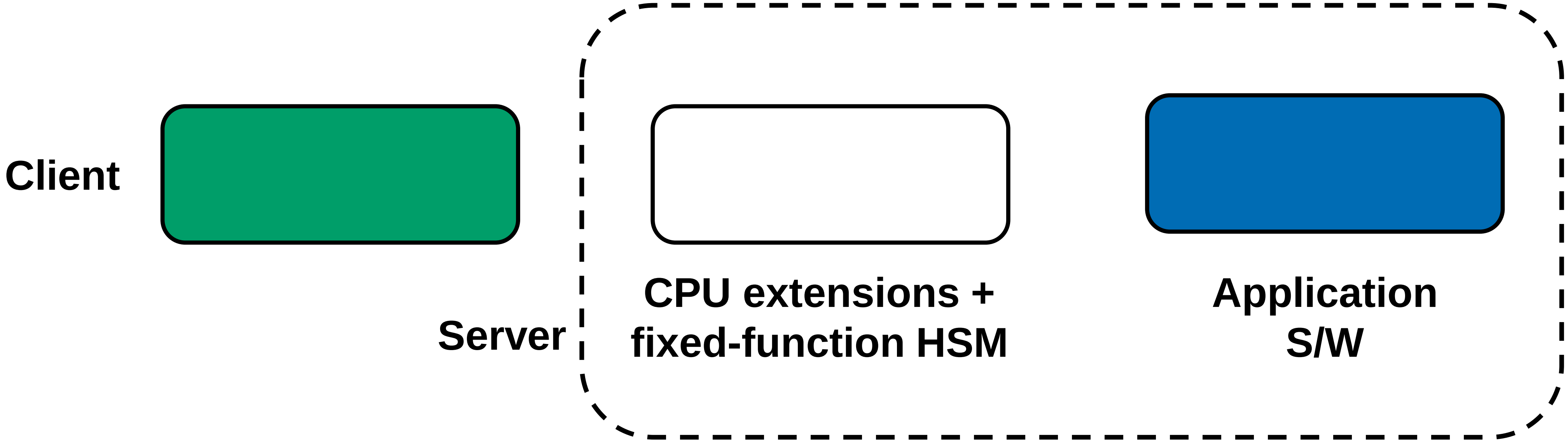
Combine with attestable HSM to assure clients

Remote attestation assures use of client data is subject to security policy



Combine with attestable HSM to assure clients

Remote attestation assures use of client data is **subject to security policy**



How does this taint-tracking policy work?

Taint tracking policy

Registers/memory have an associated “sensitive” bit (“Blinded“)

Ideal rule:

$\text{Blinded}(\text{output}_m) \leftarrow \exists n,m\text{: Blinded}(\text{input}_n) \land \text{Depends}(\text{output}_m \text{ on input}_n)$

Taint tracking policy

Registers/memory have an associated “sensitive” bit (“Blinded“)

Ideal rule:

$\text{Blinded}(\text{output}_m) \leftarrow \exists n,m\text{: Blinded}(\text{input}_n) \wedge \text{Depends}(\text{output}_m \text{ on input}_n)$

Goal: changes in sensitive state never affect non-sensitive state

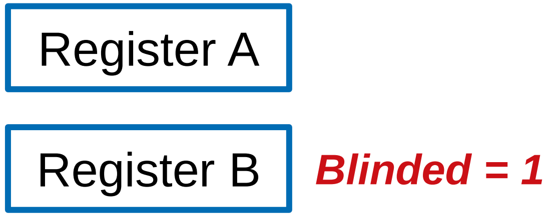
Taint tracking policy

Registers/memory have an associated “sensitive” bit (“Blinded“)

Ideal rule:

$$\text{Blinded}(\text{output}_m) \leftarrow \exists n,m\text{: Blinded}(\text{input}_n) \land \text{Depends}(\text{output}_m \text{ on input}_n)$$

Goal: changes in sensitive state never affect non-sensitive state



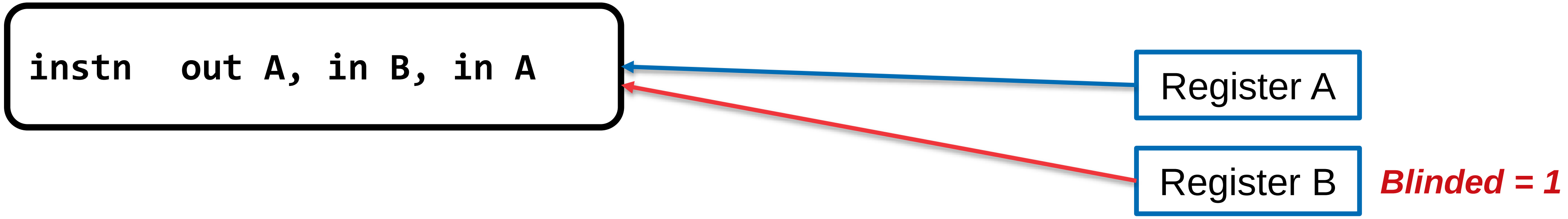
Taint tracking policy

Registers/memory have an associated “sensitive” bit (“Blinded“)

Ideal rule:

$\text{Blinded}(\text{output}_m) \leftarrow \exists n,m\text{: Blinded}(\text{input}_n) \wedge \text{Depends}(\text{output}_m \text{ on input}_n)$

Goal: changes in sensitive state never affect non-sensitive state



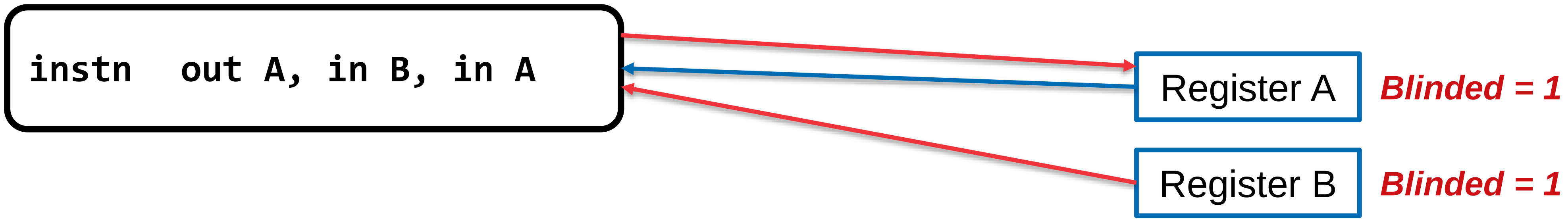
Taint tracking policy

Registers/memory have an associated “sensitive” bit (“Blinded“)

Ideal rule:

$\text{Blinded}(\text{output}_m) \leftarrow \exists n,m\text{: Blinded}(\text{input}_n) \land \text{Depends}(\text{output}_m \text{ on input}_n)$

Goal: changes in sensitive state never affect non-sensitive state



Taint tracking policy

Registers/memory have an associated "sensitive" bit ("Blinded")

Ideal rule:

$$\text{Blinded}(\text{output}_m) \leftarrow \exists n,m: \text{Blinded}(\text{input}_n) \wedge \text{Depends}(\text{output}_m \text{ on } \text{input}_n)$$

Goal: changes in sensitive state never affect non-sensitive state

```
instn   out A, in B, in A
```

```
instn   out A
```

Register A *Blinded = 1*

Register B *Blinded = 1*

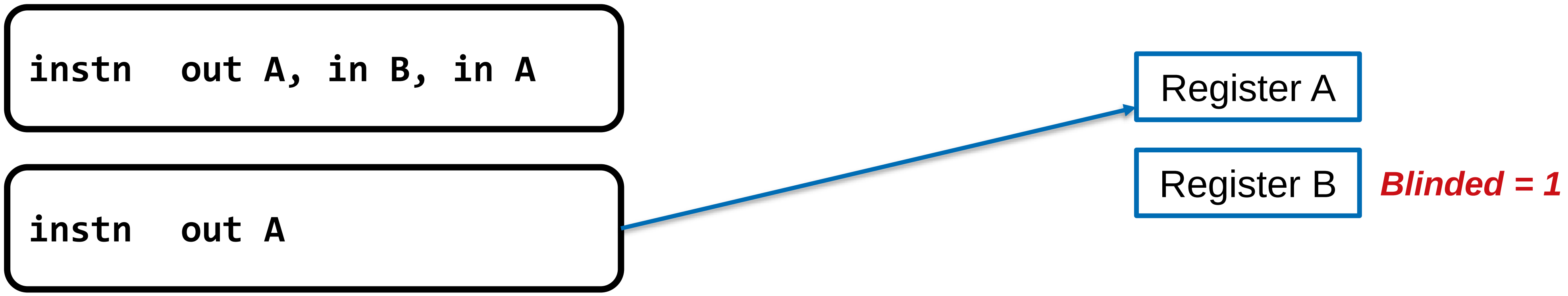
Taint tracking policy

Registers/memory have an associated "sensitive" bit ("Blinded")

Ideal rule:

$$\text{Blinded}(\text{output}_m) \leftarrow \exists n,m: \text{Blinded}(\text{input}_n) \land \text{Depends}(\text{output}_m \text{ on } \text{input}_n)$$

Goal: changes in sensitive state never affect non-sensitive state



Thinking beyond registers and memory

Taint-propagation rule must consider many different observable outputs

- Registers
- Memory values

Thinking beyond registers and memory

Taint-propagation rule must consider many different observable outputs

- Registers
- Memory values
- Control flow

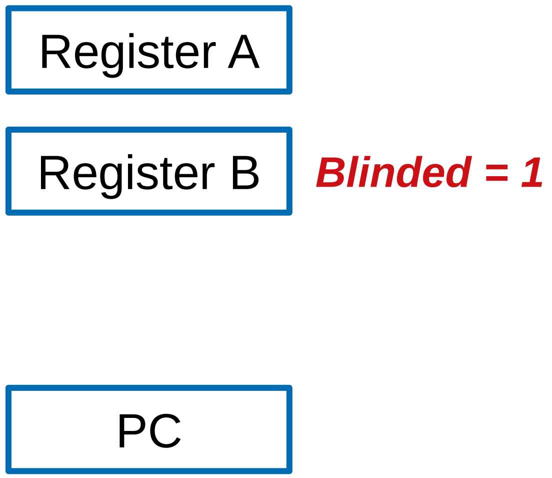
Taint tracking policy

Registers/memory have an associated “sensitive” bit (“Blinded“)

Ideal rule:

$\text{Blinded}(\text{output}_m) \leftarrow \exists n,m\text{: Blinded}(\text{input}_n) \wedge \text{Depends}(\text{output}_m \text{ on input}_n)$

Goal: changes in sensitive state never affect non-sensitive state




```
jmp     in B
```

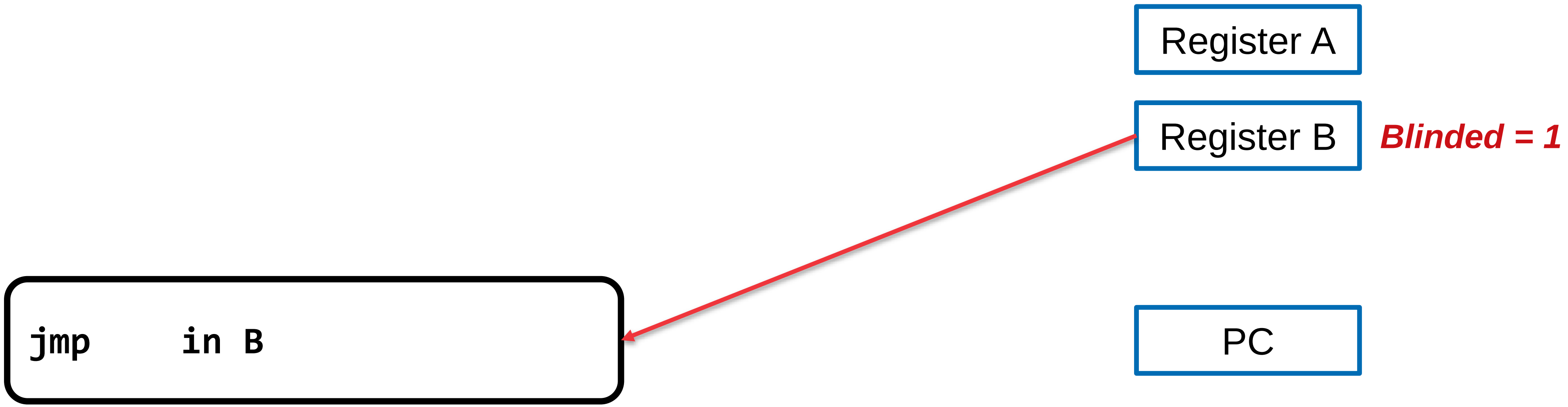
Taint tracking policy

Registers/memory have an associated “sensitive” bit (“Blinded“)

Ideal rule:

$$\text{Blinded}(\text{output}_m) \leftarrow \exists n,m\text{: Blinded}(\text{input}_n) \land \text{Depends}(\text{output}_m \text{ on input}_n)$$

Goal: changes in sensitive state never affect non-sensitive state



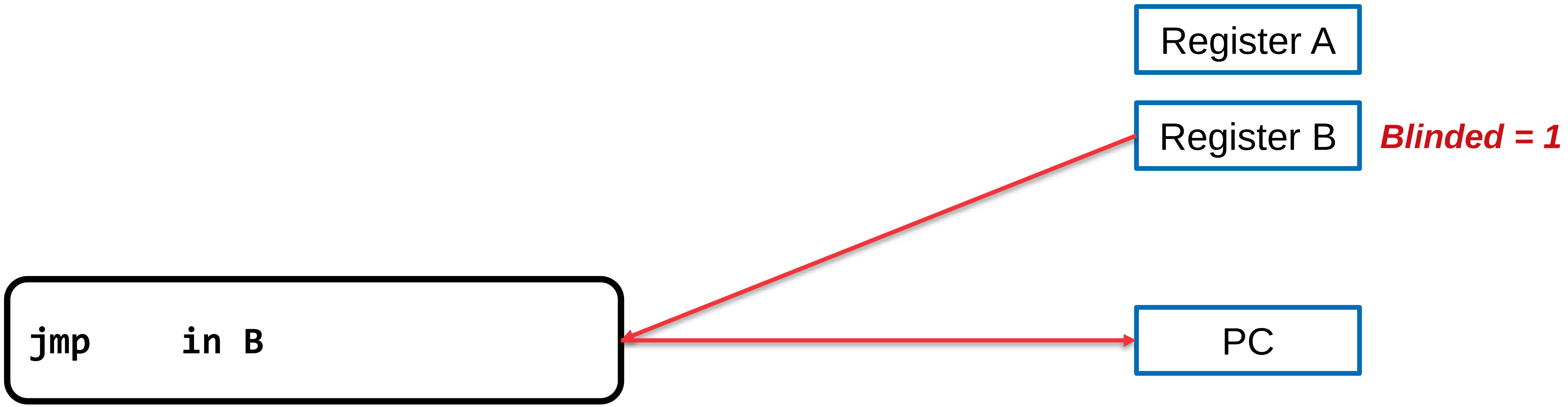
Taint tracking policy

Registers/memory have an associated “sensitive” bit (“Blinded“)

Ideal rule:

$\text{Blinded}(\text{output}_m) \leftarrow \exists n,m: \text{Blinded}(\text{input}_n) \land \text{Depends}(\text{output}_m \text{ on } \text{input}_n)$

Goal: changes in sensitive state never affect non-sensitive state



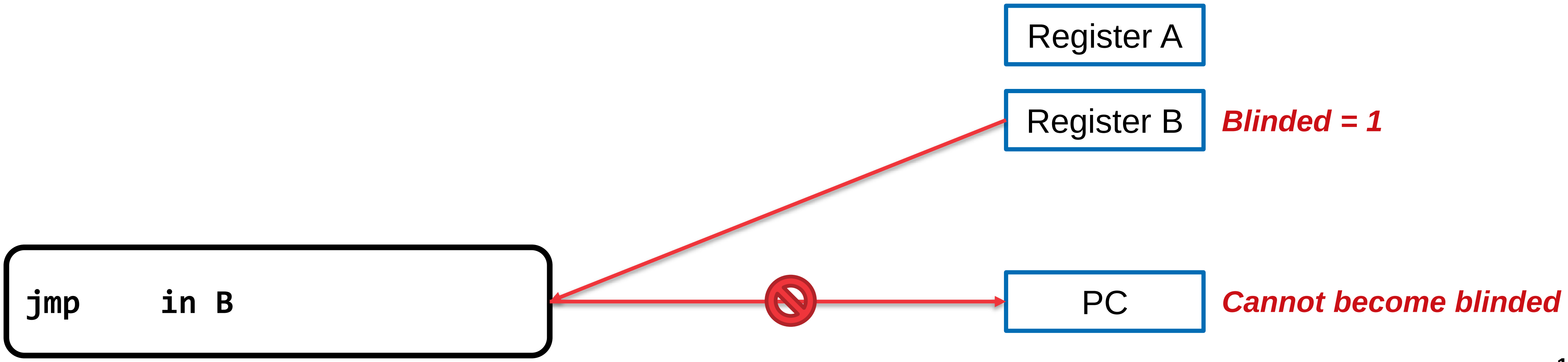
Taint tracking policy

Registers/memory have an associated “sensitive” bit (“Blinded“)

Ideal rule:

$\text{Blinded}(\text{output}_m) \leftarrow \exists n,m: \text{Blinded}(\text{input}_n) \wedge \text{Depends}(\text{output}_m \text{ on input}_n)$

Goal: **changes in sensitive state** **never affect** **non-sensitive state**



Thinking beyond registers and memory

Taint-propagation rule must consider many different observable outputs

- Registers
- Memory values
- Control flow

Thinking beyond registers and memory

Taint-propagation rule must consider many different observable outputs

- Registers
- Memory values
- Control flow
- Exceptions
- Memory access patterns

Not all of these outputs can be marked as Blinded

Thinking beyond registers and memory

Taint-propagation rule must consider many different observable outputs

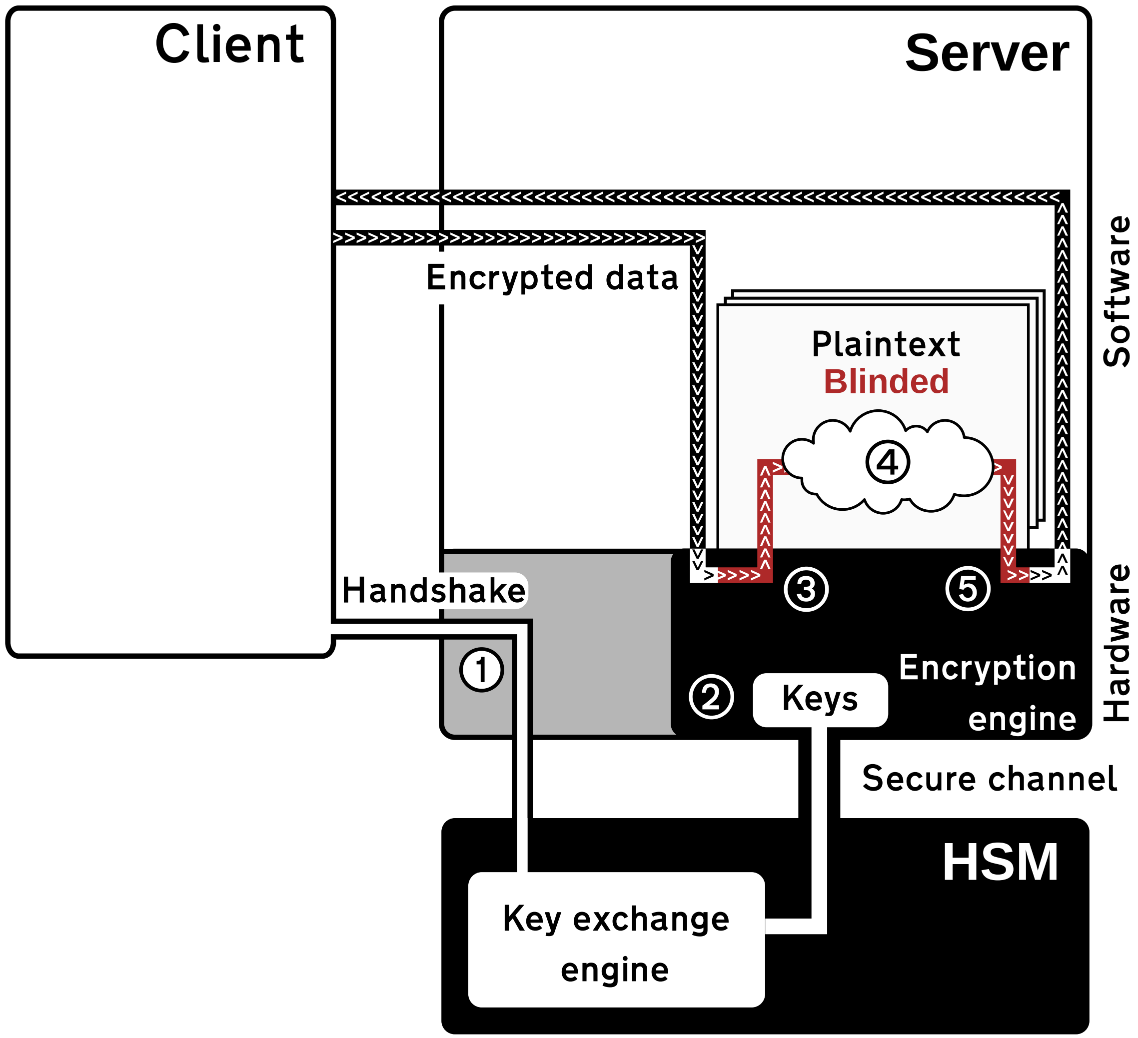
- Registers
- Memory values
- Control flow
- Exceptions
- Memory access patterns

Not all of these outputs can be marked as Blinded

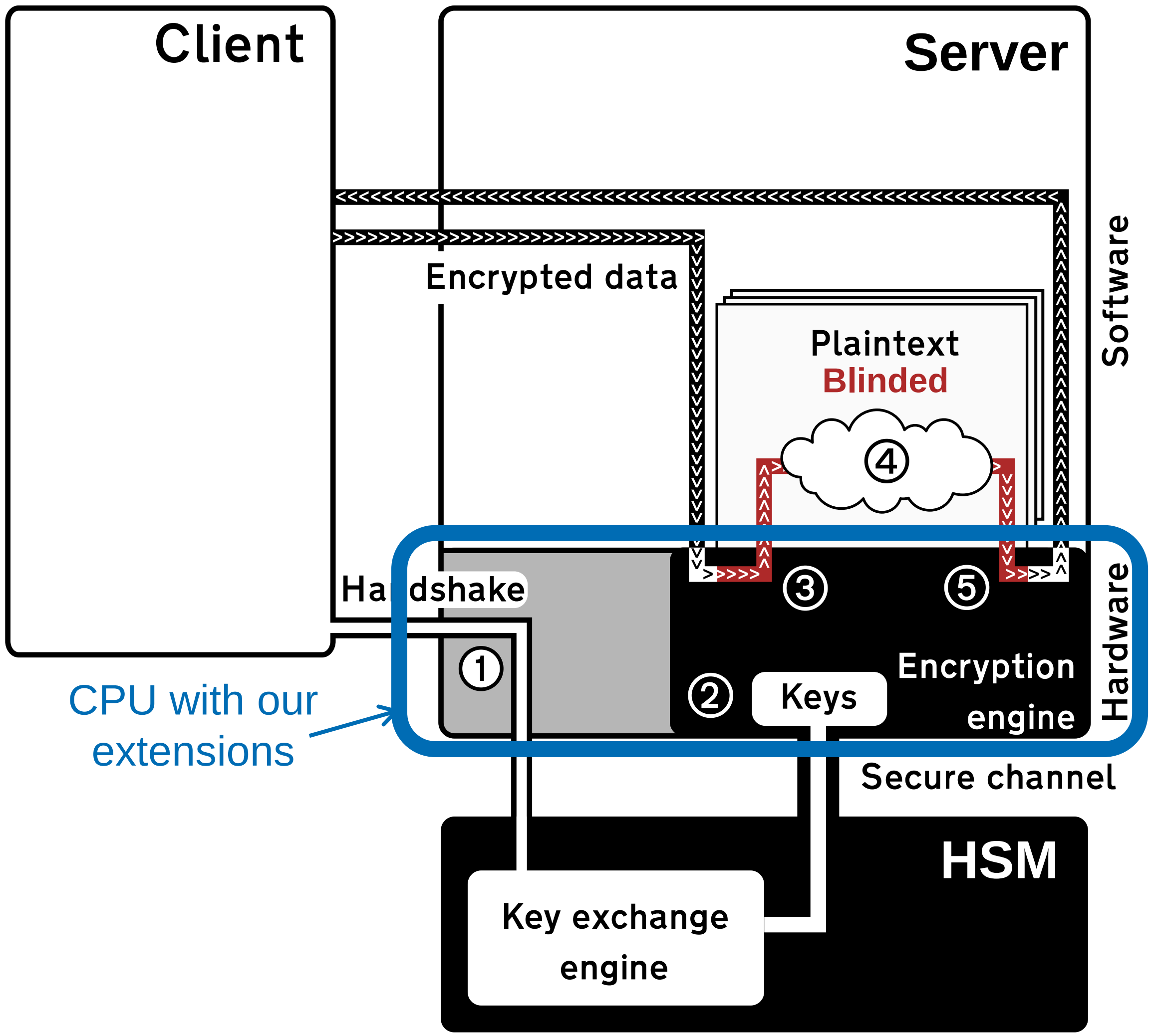
Data flows from Blinded values to “un-markable” outputs must yield a fault

Putting it all together…

BliMe Architecture

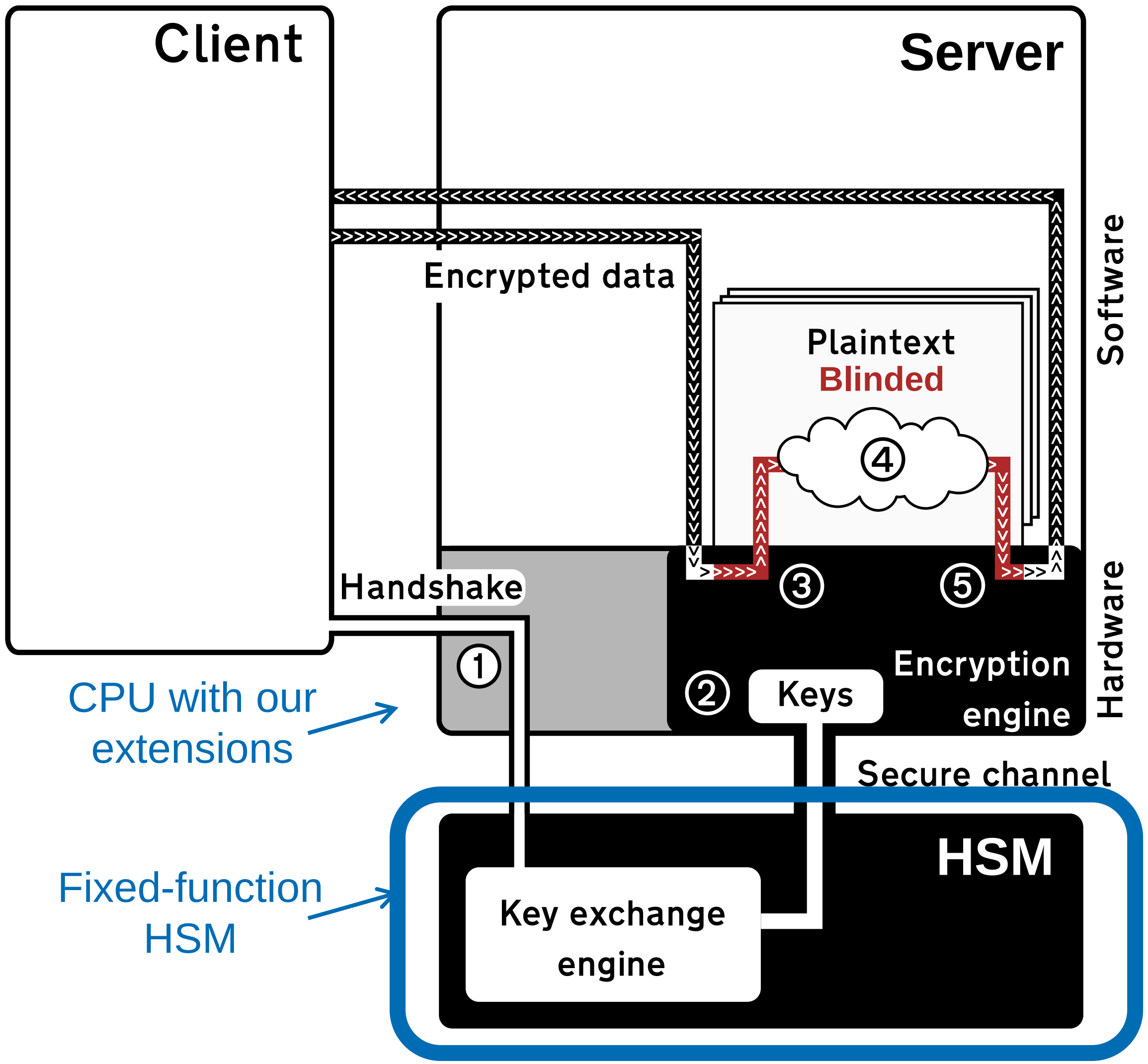


BliMe Architecture



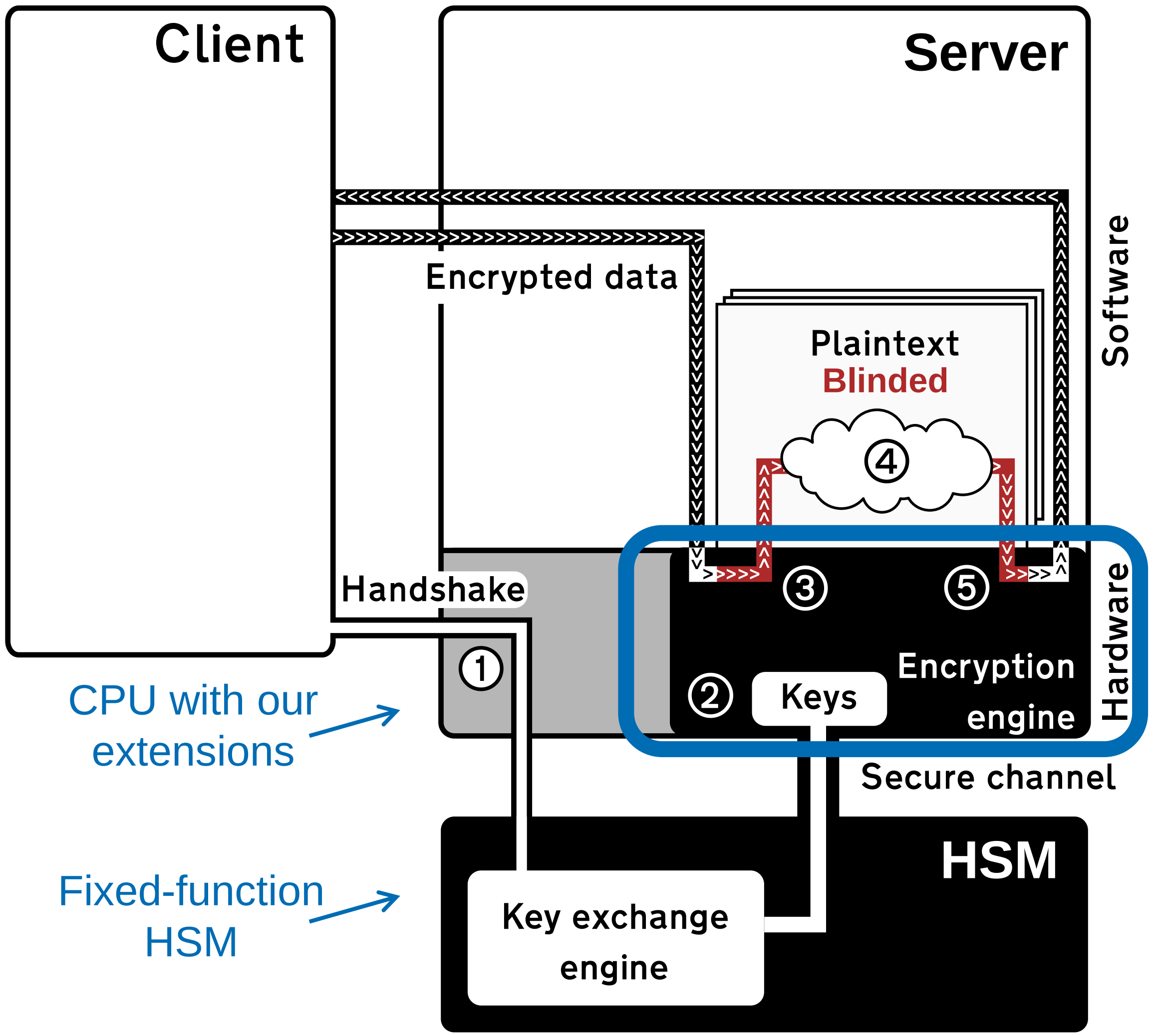
BliMe Architecture

1. **Handshake (incl. remote attestation)**



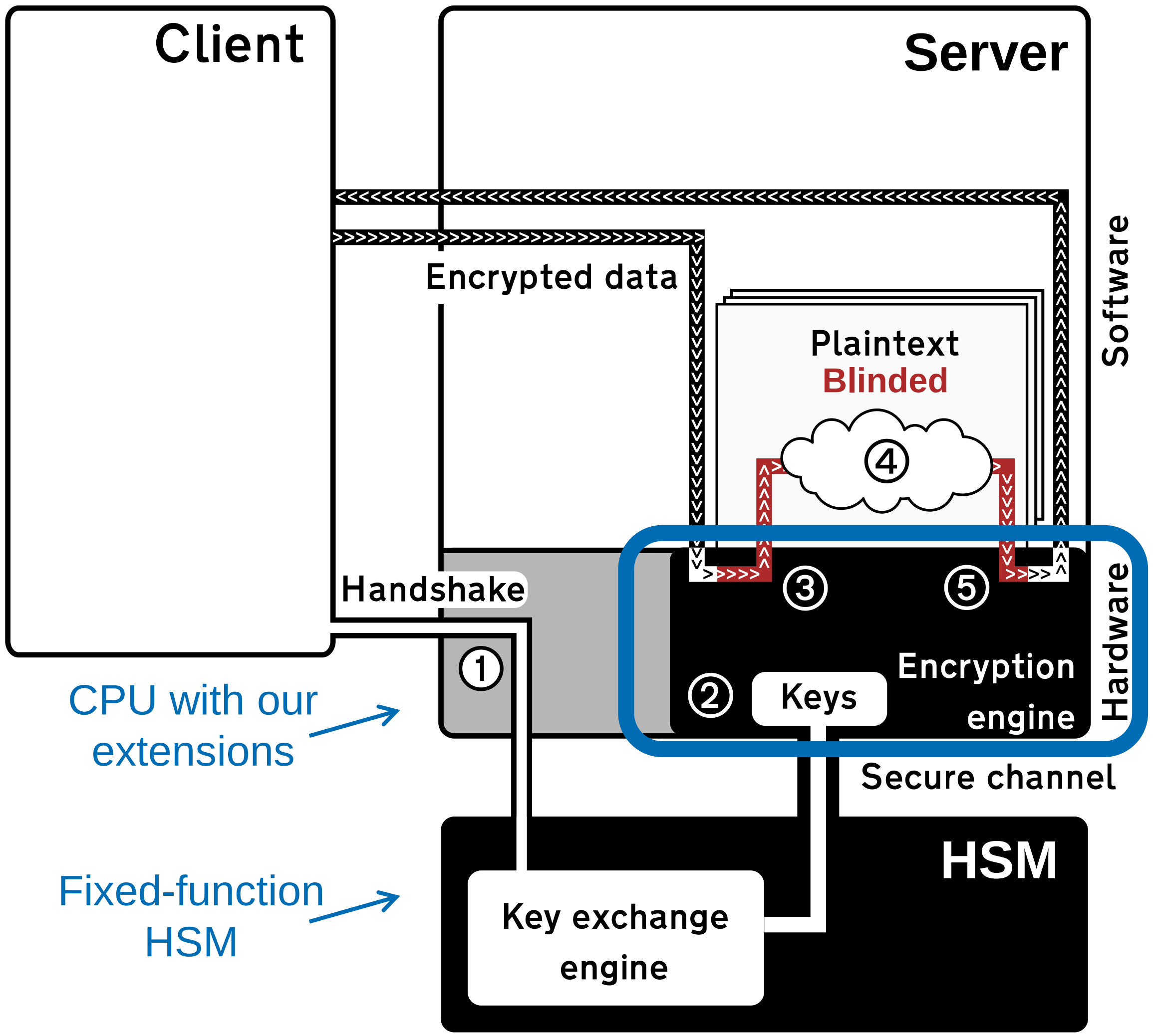
BliMe Architecture

1. **Handshake (incl. remote attestation)**
2. **Shared secret key**



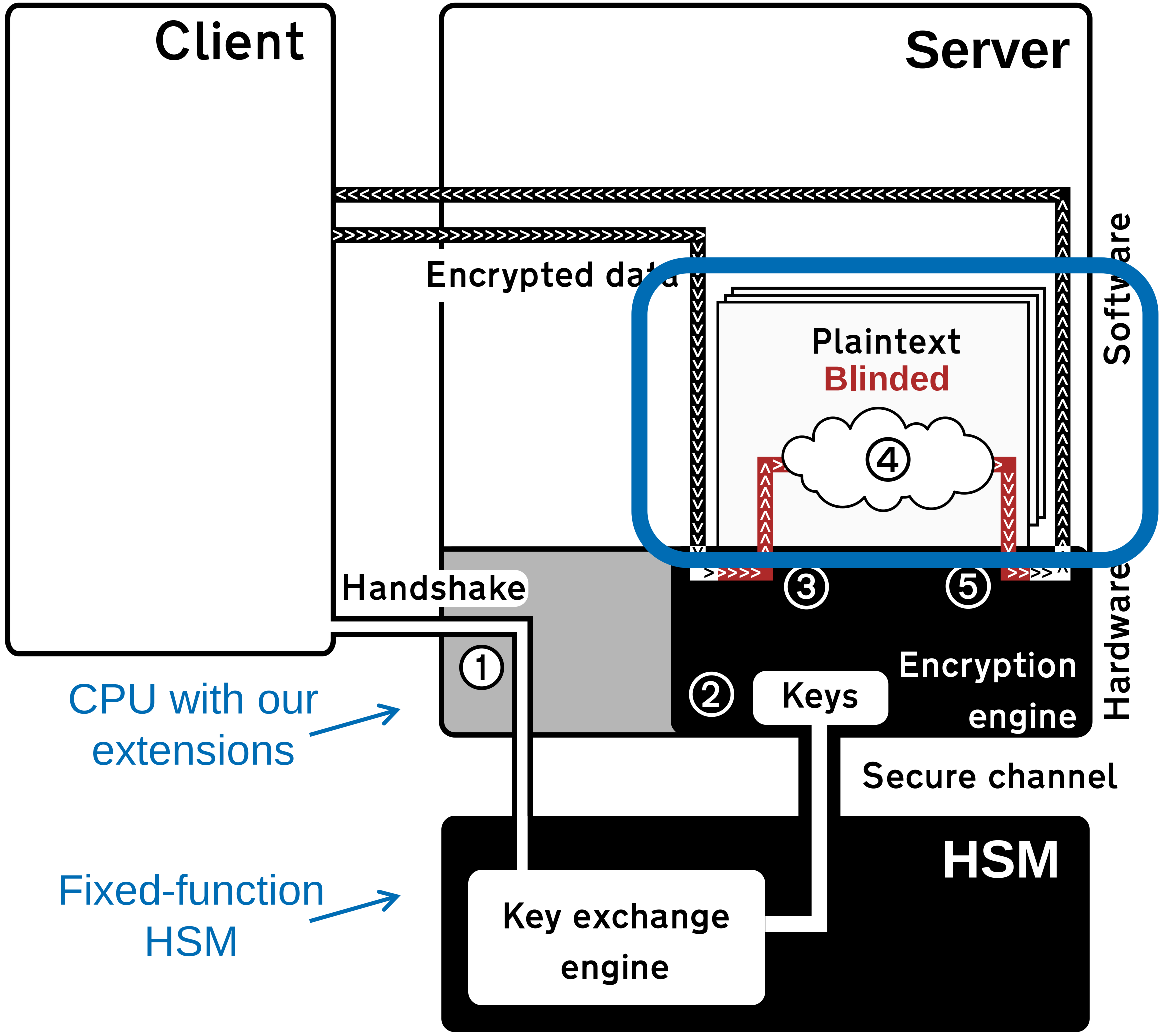
BliMe Architecture

1. **Handshake (incl. remote attestation)**
2. **Shared secret key**
3. **Atomic data import (inputs)**
   - Decrypt & blind (Blinded ← true)



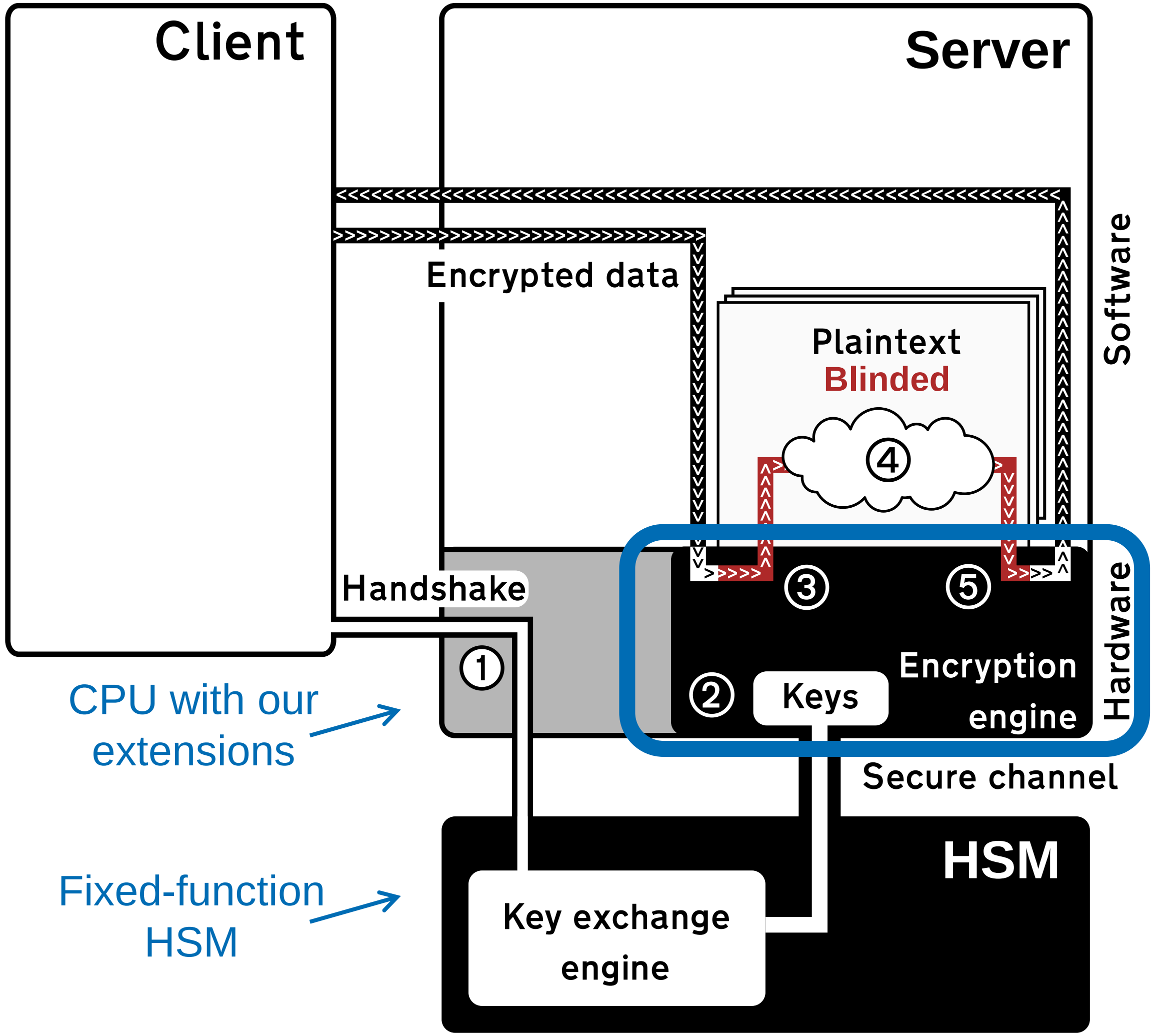
BliMe Architecture

1. **Handshake (incl. remote attestation)**
2. **Shared secret key**
3. **Atomic data import (inputs)**
   - Decrypt & blind (Blinded ← true)
4. **Safe (“blinded”) computation**
   - Enforced by BliMe HW extensions



BliMe Architecture

1. **Handshake (incl. remote attestation)**
2. **Shared secret key**
3. **Atomic data import (inputs)**
   - Decrypt & blind (Blinded ← true)
4. **Safe (“blinded”) computation**
   - Enforced by BliMe HW extensions
5. **Atomic data export (result)**
   - Encrypt & unblind (Blinded ← false)



BliMe-BOOM Implementation

On speculative OoO RISC-V BOOM core

Tagged memory: each word can be marked as blinded

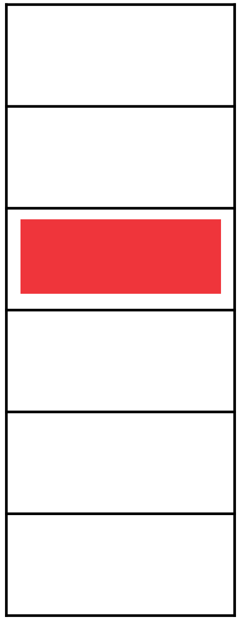
BliMe-BOOM Implementation

On speculative OoO RISC-V BOOM core

Tagged memory: each word can be marked as blinded

Instructions to mark physical memory as

- Blinded or non-Blinded



BliMe-BOOM Implementation

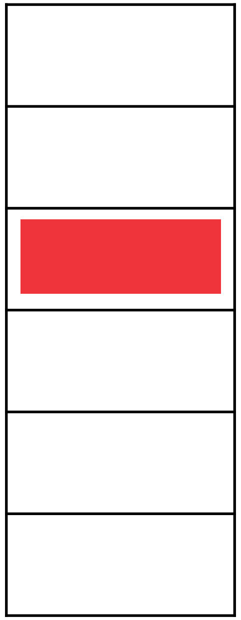
On speculative OoO RISC-V BOOM core

Tagged memory: each word can be marked as blinded

Instructions to mark physical memory as

- Blinded or non-Blinded

Implements taint-tracking for all instructions



BliMe-BOOM Implementation

On speculative OoO RISC-V BOOM core

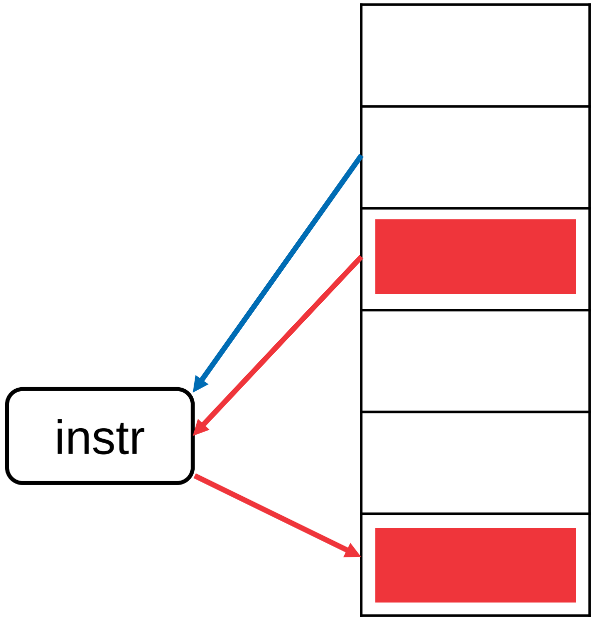
Tagged memory: each word can be marked as blinded

Instructions to mark physical memory as

- Blinded or non-Blinded

Implements taint-tracking for all instructions

- Blinded(outputs) ← Blinded($input_1$) ∨ Blinded($input_2$) ∨ …



Speculative out-of-order execution

Same security policy enforced during speculation

Instructions causing side-channel leakage (even speculatively) will fault

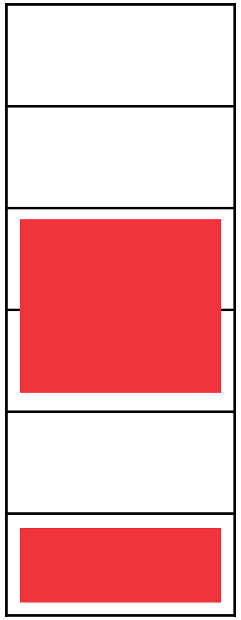
Blindedness must be tracked throughout the processor microarchitecture

- Registers, load/store queue entries, line fill buffers, etc.
- Ensured by Chisel RTL type system

Handling multiple clients simultaneously

So far, one Blinded bit for many clients

- Server can send sensitive data to the wrong client



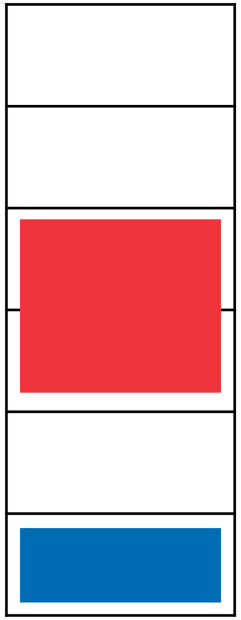
Handling multiple clients simultaneously

So far, one Blinded bit for many clients

- Server can send sensitive data to the wrong client

We need a separate sensitivity domain for each client

- Prevent clients accessing each other’s sensitive data
- Keys need to be swapped in and out for each client



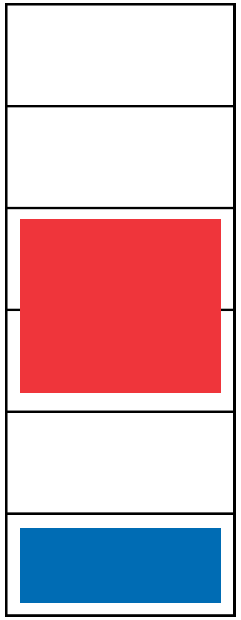
Handling multiple clients simultaneously

So far, one Blinded bit for many clients

- Server can send sensitive data to the wrong client

We need a separate sensitivity domain for each client

- Prevent clients accessing each other’s sensitive data
- Keys need to be swapped in and out for each client



Solution: Hardware support

- Hardware keeps track of sensitivity domains: multibit Blindedness tag
- Secure despite malicious OS

Evaluation

Compatibility: Tested with side-channel-resistant crypto library (TweetNaCl)

- Side-channel-resistant crypto runs without modifications

Overheads:

	Type	Δ
FPGA	LUTs & Registers	+9.0%
FPGA	Power	+1.4%
gem5	Performance (SPEC17)	+8%

Security: Formal verification in F*

Goal: changes in blinded state never affect non-blinded state

```
(**************************************************************************
 * Equivalence-based safety.
 *
 * We define safety in this case to be that the system is safe if executing on
 * equivalent (and so indistinguishable) states always results in equivalent
 * output states.
 **************************************************************************)
let equivalent_inputs_yield_equivalent_states (exec:execution_unit) (pre1 pre2 : systemState) =
    equiv_system pre1 pre2 ⇒ equiv_system (step exec pre1) (step exec pre2)


let is_safe (exec:execution_unit) =
    ∀ (pre1 pre2 : systemState). equivalent_inputs_yield_equivalent_states exec pre1 pre2
```

https://blinded-computation.github.io/blime-model/

Generating compliant code with LLVM

Problem: software might not run as-is

- BliMe hardware extensions will abort non-compliant code

TensorFlow Lite hand-ported to run on BliMe

Creating compliant code by hand is error prone

- High-level verification often insufficient
- Challenge exacerbated due to obtuse compiler behavior
- Usability/deployability challenge, not security

Challenge: solutions like Constantine[B+21] are not applicable as-is

- Uses dynamic profiling; under-approximates taint (best-effort approach)

[B+21] "Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization", ACM CCS (2021)

Generating compliant code with LLVM

Problem: software might not run as-is

- BliMe hardware extensions will abort non-compliant code

TensorFlow Lite hand-ported to run on BliMe

Creating compliant code by hand is error prone

- High-level verification often insufficient
- Challenge exacerbated due to obtuse compiler behavior
- Usability/deployability challenge, not security

Challenge: solutions like Constantine[B+21] are not applicable as-is

- Uses dynamic profiling; under-approximates taint (best-effort approach)

Ongoing work

[B+21] "Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization", ACM CCS (2021)

Summary

BliMe provides FHE-style security, but efficiently

Safely run untrusted code on sensitive data

Implemented for BOOM (speculative OoO CPU core)

Ongoing work: compiler support for usability

Paper, source code, formal model

ssg-research.github.io/platsec/blime/

How to deal with exceptions

Examples of data-dependent exceptions:

- Division by zero
- Floating-point exceptions
- …

Instructions must not raise an exception based on data-dependent conditions

Solutions:

- Unconditional faults (i.e., division by sensitive values always fails)
- Set a sensitive error flag and continue computation

Handling multiple clients simultaneously

Solution 1: BliMe-BOOM-1 + Isolation by honest-but-curious server OS

- OS keeps track of sensitivity domains
- Requires only single Blinded bit from HW: low memory overhead
- Rely on remote attestation of the entire OS to convince client

Solution 2: BliMe-BOOM-N -- Hardware support for multiple clients

- Hardware keeps track of sensitivity domains: multibit Blindedness tag
- Secure despite malicious OS
- Needs extra memory/logic to keep track of domain identifier for each granule

Generating compliant code with LLVM: our solution

Solution: Use static analysis to propagate taint

- Trade-off: over-approximation

Use SVF[S+16] as a starting point

SVF provides static value-flow graph

- Shows value dependencies within program

Identify and transform potential violations

- Apply data- and control-flow linearization

[S+16] "SVF: interprocedural static value-flow analysis in LLVM", ACM International Conference on Compiler Construction (2016)

Control-flow linearization

Control-flow decisions can leak data

- Timing, cache, branch predictor side channels

```
if (secret) {          // affects branch predictor
    arr[0] = X;        // affects cache
} else {
    arr[1] = X;        // affects cache
}
```

↓

```
taken = secret;
// if block always executed
old = arr[0];
arr[0] = (taken ? X : old);
// else block always executed
old = arr[1];
arr[1] = (!taken ? X : old);
```

Linearization allows “branching” code

- Executes all branches but keeps only desired results

Data-flow linearization

Memory accesses can leak information

- Secret-dependent memory access can leak information through side-channels

Linearization removes data-dependence

- Always access each cache line
- stride = cacheLineSize