Blinded Memory

N. Asokan, Secure Systems Group

https://asokan.org/asokan/
@nasokan

(Joint work with Hossam ElAtali, Lachlan J. Gunn, Hans Liljestrand)
This talk in a nutshell

1. Outsourced computing is everywhere…
   • Machine learning models kept behind remote APIs

2. …but this introduces security risks…
   • Providers don’t expose models/code to clients
   • Clients expose sensitive data to providers
   • Existing solutions like FHE/TEEs have drawbacks

3. …so we propose a new solution, Blinded Memory
   • Attestation + standard encryption + hardware-assisted taint tracking
   • Sensitive data not exposed to output devices or covert channels
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
Hardware-assisted TEEs are pervasive

Hardware support for
- Isolated execution: Isolated Execution Environment
- Protected storage: Sealing
- Ability to convince remote verifiers: (Remote) Attestation

**Trusted Execution Environments (TEEs)**
Operating in parallel with “rich execution environments” (REEs)

**Other Software**
**Trusted Software**
**Protected Storage**

**Root of Trust**

- Cryptocards
- Trusted Platform Modules
- ARM TrustZone
- Intel Software Guard Extensions


TEEs as an idea date back to the 1980s

1982 Texas Instruments, Guttag US4521853A
“Secure microprocessor/microcomputer with secured memory”

1996 Intertrust, Ginter et al US 5892900A
“Systems And Methods For Secure Transaction Management And Electronic Rights Protection”

1982 Texas Instruments, Guttag and Nussarallah US4521853A
“Security bit for designating the security status of information stored in a nonvolatile memory”

2002 Nokia, Kiiveri and Paatero US9111097B2
“Secure execution architecture”

2003 Texas Instruments OMAP 161x and 73x processors

2004 ARM TrustZone
Deployment of mobile TEEs date back to the 2000s

First deployment: Nokia 6630 ("Charlie")
- first 3G phone with TI OMAP 1710 processor (June 2004)

ARM TrustZone currently widely deployed
- TrustZone-M for Cortex-M class microcontrollers (2016)

Ca. 2008, TEE unheard of in academic circles
- first papers in FC 2008, ASIACCS 2009
  - [KEAR09] On-board credentials with open provisioning, ACM ASIACCS (2009)

Intel SGX
- SkyLake (2015); wide availability of SDK “democratized” TEE research
More on the history of TEEs


Protection provided by TEEs comes with caveats

TEEes provide an isolated environment for execution of software

TEEes 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?

Low performance → Low confidentiality

High performance → High confidentiality

Vanilla outsourced computing + current TEEs

Fully-Homomorphic Encryption

Our goal
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 processor (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
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 (formally verified)

instn out A, in B, in A

instn out A

jmp in B

Register A

Blinded = 1

Register B

Blinded = 1

PC

Cannot become blinded
Thinking beyond registers and memory

Taint-propagation rule must consider many different observable outputs
• Registers
• Memory values
• Memory access patterns
• Control flow
• Exceptions

Not all of these outputs can be marked as sensitive

Data flows from sensitive values to “un-markable” outputs must yield a fault
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
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 core BOOM

Tagged memory: each byte can be marked as blinded
Instructions to mark physical memory as
  • Blinded or non-Blinded

Implements taint-tracking for all instructions
  • Ideal rule: Blinded(output_m) \leftarrow \exists n, m: \text{Blinded}(\text{input}_n) \land \text{Depends}(\text{output}_m \text{ on } \text{input}_n)
  Approx. to: Blinded(outputs) \leftarrow \text{Blinded}(\text{input}_1) \lor \text{Blinded}(\text{input}_2) \lor ...
Encryption Engine

Encryption engine uses the RoCC accelerator interface in BOOM

- RoCC exposes custom logic as instructions
Handling multiple clients simultaneously

Problem: 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

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
BliMe-BOOM-N Implementation

BOOM RTL

Data tagged with **client-specific tag**

1 tag per granule

Tag size = 8 bits, granule size = 8 bytes

Future work:
  * parameterize tag size and granule size
Evaluation

Compatibility: Tested with side-channel-resistant crypto library (TweetNaCl)
- Side-channel-resistant crypto runs without modifications

Overheads

<table>
<thead>
<tr>
<th>Type</th>
<th>BOOM-1</th>
<th>BOOM-8</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUTs &amp; Registers</td>
<td>+4.0%</td>
<td>+9.0%</td>
</tr>
<tr>
<td>Power</td>
<td>+0.9%</td>
<td>+1.4%</td>
</tr>
<tr>
<td>Max clock frequency</td>
<td>No reduction</td>
<td>No reduction</td>
</tr>
</tbody>
</table>

Performance: SPEC2017

<table>
<thead>
<tr>
<th>Type</th>
<th>BOOM-1</th>
<th>BOOM-8</th>
</tr>
</thead>
<tbody>
<tr>
<td>+23%</td>
<td></td>
<td>+23%</td>
</tr>
</tbody>
</table>
BliMe-gem5 optimization

BliMe-BOOM uses **same memory request size** for data and tags

Using correct request size (1/8\textsuperscript{th}) needs **extensive changes** to baseline

**Solution:** Use **gem5 simulator** to perform evaluation with **correct size**
- BliMe-gem5-optimized

Could Δ in performance just be caused by moving to gem5?
- Implement BliMe-gem5 with BliMe-BOOM configs
- BliMe-gem5 matches BliMe-BOOM in average performance (SPEC 2017)
Evaluation

Compatibility: Tested with side-channel-resistant crypto library (TweetNaCl)
• Side-channel-resistant crypto runs without modifications

Overheads

<table>
<thead>
<tr>
<th>Type</th>
<th>BOOM-1</th>
<th>BOOM-8</th>
<th>FPGA</th>
<th>Performance: SPEC2017</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUTs &amp; Registers</td>
<td>+4.0%</td>
<td>+9.0%</td>
<td>+4.0%</td>
<td>+23%</td>
</tr>
<tr>
<td>Power</td>
<td>+0.9%</td>
<td>+1.4%</td>
<td>+0.9%</td>
<td>+23%</td>
</tr>
<tr>
<td>Max clock frequency</td>
<td>No reduction</td>
<td>No reduction</td>
<td>No reduction</td>
<td>+25%</td>
</tr>
</tbody>
</table>

Performance: 8% average overhead on gem5 after optimization
**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.
******************************************************************************

```ocaml
let equivalent_inputs_yield_equivalent_states (exec:execution_unit) (pre1 pre2 : systemState) =
  equiv_system pre1 pre2 ⇒
  equiv_system (step exec pre1) (step exec pre2)
```

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

[https://blinded-computation.github.io/blime-model/](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

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\cite{B+21} are not applicable as-is
• Uses dynamic profiling; under-approximates taint (best-effort approach)

\cite{B+21} "Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization", ACM CCS (2021)
Generating compliant code with LLVM: our solution

Solution: Use static analysis to propagate taint
  • Trade-off: over-approximation

Use SVF\textsuperscript{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

\textsuperscript{S+16} "SVF: interprocedural static value-flow analysis in LLVM", ACM International Conference on Compiler Construction (2016)
Adapting TensorFlow Lite to BliMe

Compiled image classification example

Some manual fixes required in TensorFlow Lite library source code
  • e.g., array access expansion for softmax lookup table

In progress:
  • For TensorFlow Lite: try more example models
  • For the compiler
    • Ensure soundness
    • Produce warnings for untransformed libraries
In Progress: BliMe\textsuperscript{NG}

**HW accelerators common in outsourced computation**

- Customized to application

**ML accelerators useful for all ML workloads**

**Goal: Adapt BliMe to ML accelerator framework**

**Prominent open-source frameworks:**

- NVDLA (by NVIDIA)
- Gemmini (by BOOM team)
In progress: summary

Compiler support: improving usability/deployability

Hardware improvements:
  Implementing tag cache
  Adapting BliMe to Gemmini accelerator

Evaluation: Experimenting with more TensorFlow models on BliMe
Summary

BliMe provides FHE-style security, but efficiently

Server can safely run untrusted code on sensitive data

Incorporated into speculative OoO RISC-V core BOOM

In progress: compiler support, tag cache, TensorFlow, Gemmini

Paper, source code etc. at https://ssg-research.github.io/blime/

Summary

BliMe provides FHE-style security, but **efficiently**

Server can **safely run** untrusted code on sensitive data

Incorporated into speculative OoO RISC-V core **BOOM**

In progress: compiler support, tag cache, TensorFlow, Gemmini

Paper, source code etc. at [https://ssg-research.github.io/blime/](https://ssg-research.github.io/blime/)

