Language-Based Control and Mitigation of Timing Channels

Danfeng Zhang, Aslan Askarov, Andrew C. Myers
Cornell          Harvard          Cornell

PLDI 2012
Timing Channels

Hard to detect and prevent
Timing Channels: Examples

• Network timing attacks
  – RSA keys leaked by encryption time [Brumley&Boneh 05], measured across network
  – Login/load time reveals validity of usernames, login status, size and contents of shopping cart [Bortz&Boneh 07]

• Cache timing attacks
  – AES keys leaked in almost real time by timing memory accesses (only 100 encryptions!) [Gullasch et al. 11]

• Covert timing channels
  – Transmit confidential data by controlling response time, e.g., combined with SQL injection [Meer&Slaviero 07]

Timing channels are real threats to security!
Previous Work

- language-based security
- system-level mitigation
- secure architecture
- timing channel mitigation
This Work

• Abstraction bridges different levels of mitigation

• Formally proved security guarantees

• Implemented/evaluated on real-world code
**Security Model**

- **Security policy lattice**
  - Information has *label* describing intended confidentiality
  - In general, the labels form a *lattice*
  - For this talk, a simple lattice:
    - **S** (secret) – High
    - **P** (public) – Low

- **Attacker model (at level P in the talk)**
  - Sees contents of public memory (storage channel)
  - Sees timing of updates to public memory (timing channel)
    - A real threat for cloud computing ...
A Subtle Example

1 if (secret1)
2 secret2 := public1;
3 else
4 secret2 := public2;
5 public3 := public1;

The data cache affects timing

Programming model does not capture timing!
```java
if (secret1) {
    secret2 := public1;
} else {
    secret2 := public2;
    public3 := public1;
}
```

Beneath the Language Abstraction

concise and sufficient interface?
A Language-Level Abstraction

\[(x := e)\]

**Machine environment**: state affecting timing but invisible at language level
A Language-Level Abstraction

Each operation has **read label, write label** governing interaction with machine environment

\[(x := e)[\ell_r, \ell_w]\]

machine environment

logically partitioned by security level
(e.g. public part vs. secret part of cache)
Read Labels

\[(x := e)_{[\ell_r, \ell_w]}\]

- Restricts how machine environment affects timing
- **Upper bound** on timing influence
  - e.g. secret part cannot affect timing when read label is \(P\)

\[(s_1 := s_2)_{[P, \ell_w]}\]
Write Labels

\((x := e)_{[\ell_r, \ell_w]}\)

- Restricts how machine environment is modified
- **Lower bound** on updates to machine environment
  - e.g. no updates to public part when write label is \(S\)

\((s_1 := s_2)_{[\ell_r, S]}\)

\[
\begin{array}{c}
\text{S} \\
\text{P}
\end{array}
\quad = \quad
\begin{array}{c}
\text{S'} \\
\text{P}
\end{array}
\]

after one step
Security Properties

- The language implementation must satisfy three (formal) properties in the form of operational semantics
  - Read label property
  - Write label property
  - Single-step machine environment noninterference: no leaks from secret part to public part in one step

All requirements are realizable on commodity HW!
We simulated a more efficient implementation
Impact of Properties

Guidance to designers of future secure architectures

• **Security**
  – Possible to verify architecture design
    (informally verified two designs in paper)

• **Performance**
  – Caching secret data in secret cache [Wang et. al. 07] is (sometimes) unnecessary
Overview

• Background and motivation
• A language-level timing abstraction
• Type system and security guarantees
• Evaluation
Language

e ::= n | x | e op e

c ::= skip_{lr,lw} | (x := e)_{lr,lw} | c;c | (while e do c)_{lr,lw}
   | (if e then c_1 else c_2)_{lr,lw}
   | (mitigate_{eta} (e, l) c)_{lr,lw} | (sleep e)_{lr,lw}

discuss later
Type System

• Typing rules

\[ \Gamma, pc, \tau \vdash c : \tau' \]

\textit{Meaning:} (in context $\Gamma, pc$)

begin time of $c$ contains information of (at most) label $\tau$

$\Rightarrow$ end time contains information of (at most) label $\tau'$
Rule for Assignment

\[ \Gamma \vdash e : \ell \quad pc \sqsubseteq \ell_w \quad \ell \cup pc \cup \tau \cup \ell_r \sqsubseteq \Gamma(x) \]

\[ \Gamma, pc, \tau \vdash x := e_{[\ell_r, \ell_w]} : \Gamma(x) \]

Theorem: a well-typed program without mitigate leaks nothing via timing channels
Language-Level Mitigation

Disallowing all leakage is too restrictive

Assume the validity of username is secret:

```
mitigate(1,S) {
  (name, pass) := input
  if (exists(name))
    check(name, pass);
  public := 1;
}
```

```
(name, pass) := input
if (exists(name))
  check(name, pass);
public := 1;
```
Quantitative Result

\[
\text{mitigate}(e, \ell) \{c\}
\]

Idea: execute \( c \), but using dynamic mitigation to bound timing channel leakage

Theorem: a well-typed program has bounded leakage of \( \log | \{T_1, T_2, T_3, \ldots \} | \).
Bridging System-Level Mitigation

A well-typed program has bounded leakage of \( \log | \{ T_1, T_2, T_3, \ldots \} | \).

Predictive mitigation [Askarov et al. 10, Zhang et al. 11]

- User defined leakage bound,
  e.g. \( O(\log^2 T) \), by delaying events

A well-typed program leaks at most \( O(\log^2 T) \) bits using predictive mitigation.
Overview

• Background and motivation
• A language-level timing abstraction
• Type system and security guarantees
• Evaluation
Evaluation Setup

- Statically partitioned cache/TLB, simulated on SimpleScalar, v.3.0.e

<table>
<thead>
<tr>
<th>Name</th>
<th># of sets</th>
<th>issue</th>
<th>block size</th>
<th>latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Data Cache</td>
<td>128</td>
<td>4-way</td>
<td>32 byte</td>
<td>1 cycle</td>
</tr>
<tr>
<td>L2 Data Cache</td>
<td>1024</td>
<td>4-way</td>
<td>64 byte</td>
<td>6 cycles</td>
</tr>
<tr>
<td>L1 Inst. Cache</td>
<td>512</td>
<td>1-way</td>
<td>32 byte</td>
<td>1 cycle</td>
</tr>
<tr>
<td>L2 Inst. Cache</td>
<td>1024</td>
<td>4-way</td>
<td>64 byte</td>
<td>6 cycles</td>
</tr>
<tr>
<td>Data TLB</td>
<td>16</td>
<td>4-way</td>
<td>4KB</td>
<td>30 cycles</td>
</tr>
<tr>
<td>Instruction TLB</td>
<td>32</td>
<td>4-way</td>
<td>4KB</td>
<td>30 cycles</td>
</tr>
</tbody>
</table>
Web Login

• Learn valid usernames via timing [Bortz&Boneh, 07]
• Secret
  – Validity of (username, password) pairs
• Inputs
  – 100 different (username, password) pairs
Correctness

Timing is independent of secret valid usernames.

With mitigation:
- no mitigation
- valid usernames
- invalid usernames

Timing is independent of secret
Performance

- **nopar**: unmodified hardware
- **moff**: partitioned hardware without mitigation
- **mon**: partitioned hardware with mitigation

<table>
<thead>
<tr>
<th></th>
<th>nopar</th>
<th>moff</th>
<th>mon</th>
</tr>
</thead>
<tbody>
<tr>
<td>ave. time (valid)</td>
<td>70618</td>
<td>78610</td>
<td>86132</td>
</tr>
<tr>
<td>ave. time (invalid)</td>
<td>39593</td>
<td>43756</td>
<td>86147</td>
</tr>
<tr>
<td>overhead (valid)</td>
<td>1</td>
<td>1.11</td>
<td>1.22</td>
</tr>
</tbody>
</table>

~20% overhead
RSA

- RSA reference implementation
- Secret
  - private keys
- Inputs
  - different encrypted messages
Correctness

leakage is eliminated
Related Work

• **Language-based security** [Volpano&Smith 97, Smith 01, Agat 00, Barth et al. 06]
  – Simplistic assumption: ignore instruction cache
  – Too restrictive: disallow confidential data in branch condition

• **Secure architecture** [Wang et al. 07, Li et al. 11]
  – Too strong security requirements

• **System-level mitigation** [Kang et al. 93, 96, Köpf&Dürmuchs 09, Askarov et al. 10, Zhang et al. 11]
  – Weaker attacker model
  – Have to mitigate benign variations in timing
Conclusion

• A language-level abstraction
  – bridges three separate levels of timing-channel mitigation
  – guidance to designers of future secure architectures

• A type system with proved guarantees
  – timing channels are provably controlled
  – dynamic mitigation used to increase expressiveness

• Evaluation
  – the technique is sound and appears practical